НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____2020 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Веб-додаток для створення та адміністрування чат-кімнат на мікро сервісній архітектурі»

Виконав:
студент IV курсу, групи ІО-64
Дем'янчук Павло Степанович _____

Керівник:
Асистент кафедри ОТ.
Каплунов Артем Володимирович _____

Консультант з нормоконтролю:
Професор кафедри ОТ, д.т.н.,
Сімоненко Валерій Павлович _____

Рецензент:
_____
_____ _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____2020 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Дем'янчуку Павлу Степановичу

1. Тема проєкту «Веб-додаток для створення та адміністрування чат-кімнат на мікро сервісній архітектурі», керівник проєкту Каплунов Артем Володимирович, асистент кафедри ОТ., затверджені наказом по університету від «07» травня 2020 р. № 1081-с

2. Термін подання студентом проєкту *26 травня 2020р.*

3. Вихідні дані до проєкту *див. технічне завдання*

4. Зміст пояснювальної записки *Аналіз і характеристика об'єкта проектування, обґрунтування оптимального варіанта реалізації мети цієї роботи, розробка додатку: вибір технологій та їх обґрунтування, основні рішення з реалізації додатку. Висновки.*

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо) *приниципова схема, функціональна схема, структурна схема*

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
нормоконтроль	Сімоненко В. П., професор, д.т.н.		

7. Дата видачі завдання 01.09.2019

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	*Затвердження теми роботи*	*01.09.2019*	
2.	*Вивчення та аналіз завдання*	*15.12.2019-15.03.2020*	
3.	*Розробка архітектури додатку*	*15.03.2020-25.03.2020*	
4.	*Написання програмної частини*	*25.03.2020-05.04.2020*	
5.	*Тестування та виправлення помилок*	*05.04.2020-15.04.2020*	
6.	*Оформлення пояснювальної записки*	*15.04.2020-20.05.2020*	
7.	*Захист програмного продукту*	*25.04.2020*	
8.	*Передзахист*	*26.05.2020*	
9.	*Захист*	*18.06.2020*	

Студент Павло ДЕМ'ЯНЧУК

Керівник Артем КАПЛУНОВ

Анотація

Робота присвячена розробці веб-додатку на базі мікросервісної архітектури. Кожен компонент системи, побудованої на мікросервісній архітектурі, являє собою просту програму, що виконує одне вузьконаправлене завдання і також є елементом однієї великої системи. Мікросервіси відносно прості в створенні та підтримці, завдяки чому також популяризують принцип швидкої розробки додатків RAD (rapid application development).

Останнім часом створилась стійка тенденція до більш гнучкого стилю розробки великих систем. За допомогою мікросервісного підходу системи можуть швидко підлаштуватися під нові вимоги до програмних компонентів, уникнувши трудомісткого перепрограмування і повторного тестування, необхідного у випадку складних, монолітних додатків. Через постійне оновлення технологічної бази для вирішення незмінних задач сучасних систем, мікросервіси стануть незамінні, оскільки підприємства будуть змушені модернізовувати свій програмний арсенал.

Мікросервісний архітектурний підхід полягає у рознесенні функціоналу на самостійні підсистеми задля отримання слабкої звязності компонентів. Це дозволяє в подальшому легко доповнювати, переробляти та замінювати кожен незалежний компонент без беспосереднього втручання у роботу всієї системи.

Аннотация

Работа посвящена разработке веб-приложения на основе микросервисной архитектуры. Каждый компонент системы, построенный на микросервисной архитектуре, представляет собой простую программу, которая выполняет одну узконаправленную задачу и в то же время является элементом одной большой системы. Микросервисы относительно просты в создании и обслуживании, что также продвигает новое представление в розробке програмного обеспечения RAD (rapid application development).

Не так давно начала наблюдатся устойчивая склонность к более гибкому подходе к разработке больших систем. Благодаря розбиению на микросервисы

системы быстро адаптируються к новым запросам по разроботке программного обеспечения, избегая трудоемкого процеса перекодирования и повторного тестирования существующего функционала, что зачастую выполняется при росширении больших монолитных систем. В связи с постоянным обновлением технологической базы для решения постоянных проблем современных систем, микросервисы станут незаменимыми, так как компании будут вынуждены модернизировать свой программный арсенал.

Микросервисный архитектурный подход заключается в распределении функционала по самостоятельным подсистемам для получения слабой связности компонентов. Это позволяет легко дополнять, переделывать и заменять каждый независимый компонент без непосредственного вмешательства в работу всей системы.

Abstract

The work is devoted to the development of a web application based on microservice architecture. Each component of the system, built on microservice architecture, is a simple program that performs one narrowly focused task and at the same time is an element of one large system. Microservices are relatively easy to create and maintain, which also promotes the principle of RAD (rapid application development).

Recently, there has been a steady trend towards a more flexible style of developing large systems. With a microservice approach, systems can quickly adapt to new software requirements, avoiding the time-consuming reprogramming and re-testing required for complex, monolithic applications. Due to the constant updating of the technological base to solve the constant problems of modern systems, microservices will become indispensable, as companies will be forced to modernize their software arsenal.

The microservice architectural approach consists in distribution of functionality on independent subsystems for reception of weak connectivity of components.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проєкт	2	
2	А4	ДП 6407. 02.000 ТЗ	Технічне Завдання	4	
3	А4	ДП 6407. 03.000 ПЗ	Пояснювальна записка	62	
4	А4	ДП 6407. 04.000 Д1	Принципова схема	1	
5	А4	ДП 6407. 05.000 Д2	Функціональна схема	1	
6	А4	ДП 6407. 06.000 Д3	Структурна схема	1	
7	А4	ДП 6407. 07.000 Д4	Програмний код	11	

					ДП 6407.01.000 ВП			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив		*Дем'янчук П. С.*			*Веб-додаток для створення та адміністрування чат-кімнат на мікросервісній архітектурі* *Відомість дипломного проєкту*	Літ.	Аркуш	Аркушів
Перевір.		*Каплунов А. В.*					1	1
						НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, Група ІО-64		
Н. контр.		*Сімоненко В. П.*						
Затверд.								

Технічне завдання

до дипломного проєкту

на тему: «Веб-додаток для створення та адміністрування чат-кімнат на мікросервісній архітектурі»

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування: «Веб-додаток для створення та адміністрування чат-кімнат на мікросервісній архітектурі».

Область застосування: розроблений веб-додаток можуть використовувати будь-які користувачі у мережі інтернет для спілкування з іншими користувачами через чат-кімнати.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського дипломного проекту, затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут ім. Ігоря Сікорського».

3.МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є створення веб-додатку на базі мікросервісної архітектури, що надає можливість створювати та адмініструвати чат-кімнати для спілкування у мережі інтернет.

4.ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є науково-технічна література, публікації в спеціалізованих періодичних виданнях та публікації в мережі Інтернет по даній темі.

5.ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

Розроблений веб-додаток повинен:

- Бути відкритим для доступу в браузері

- Надавати користувачам можливість створення та адміністрування чат-кімнат
- Забезпечувати відправлення та отримання повідомлень у рамках кожної чат-кімнати

5.2. Вимоги до пристрою клієнта

- Підключення до мережі інтернет
- Наявність веб-браузеру для доступу до веб-додатку

6. ЕТАПИ РОЗРОБКИ

	Дата
Вивчення необхідної літератури	19.02.2020
Складання і узгодження технічного завдання	06.03.2020
Написання вступної частини та огляд рішень	19.03.2020
Розробка архітектури додатку	03.04.2020
Написання програмної частини	10.04.2020
Тестування та виправлення помилок	01.05.2020
Оформлення документації дипломного проекту	15.05.2020
Попередній захист та проходження нормативного контролю	29.05.2020
Захист дипломного проекту	15.06.2020

Пояснювальна записка

до дипломного проєкту

на тему: «Веб-додаток для створення та адміністрування чат-кімнат на мікросервісній архітектурі»

Київ – 2020 року

ЗМІСТ

ВСТУП

Високі вимоги до сучасних веб-додатків, такі як можливість надання програмного інтерфейсу, інтеграція з іншими веб-додатками, обробка великої кількості запитів, масштабованість, забезпечення необхідної швидкості доступу до інформації, призводять до того, що корпоративні веб-додатки з монолітною архітектурою стають незручними в подальшій розробці. Тому сьогодні моделювання архітектури програмного продукту є однозначно найважливішою частиною процесу розробки. Вибір конкретного архітектурного рішення залежить від багатьох чинників, зокрема, призначення розроблюваної системи. Останнім часом все частіше перевага віддається саме мікросервісам [1].

Актуальність теми даної роботи викликана великою кількістю архітектурних рішень, які на сьогоднішній день користуються популярністю. Часто розробники без проведення детального аналізу предметної області обирають архітектурний стиль, що не підходить систем. Неправильно обрана архітектурна база може створити додаткові складнощі при розробці програмного забезпечення, а інколи навіть поставити під сумнів подальшу розробку чи підтримку системи. Для уникнення подібних ситуацій дана робота розгляне актуальність вибору мікросервісного архітектурного стилю на прикладі розробленого веб-додатку.

Основна мета даної роботи полягає у визначенні та описі умов, необхідних для здійснення ефективного проектування та розробки веб-додатку на основі мікросервісної архітектури, а також у розгляді особливостей та обмежень, таких систем.

Робота складається зі вступу, чотирьох розділів, та висновку по проекту. У першому розділі проведено огляд наявних архітектурних рішень, у другому описані обрані для реалізації технології та основні вимоги до системи. У третьому описана реалізація всіх складових веб-додатку та його функціональні можливості. У висновках проведено підсумки по проведеній роботі.

РОЗДІЛ 1
ПРОЕКТУВАННЯ ВЕБ ДОДАТКІВ

Сучасні веб-технології дозволяють здійснювати розробку як простих онлайн-сервісів, так і складних інформаційних систем. Для успішної розробки таких програм необхідно розуміти ряд загальних процесів розробки програмного забезпечення (ПО): основні дії по розробці, які повинні бути виконані, взаємозвязок та послідовність виконання цих дій [2]. Етапи розробки програмного забезпечення зображено на рисунку 1.1.

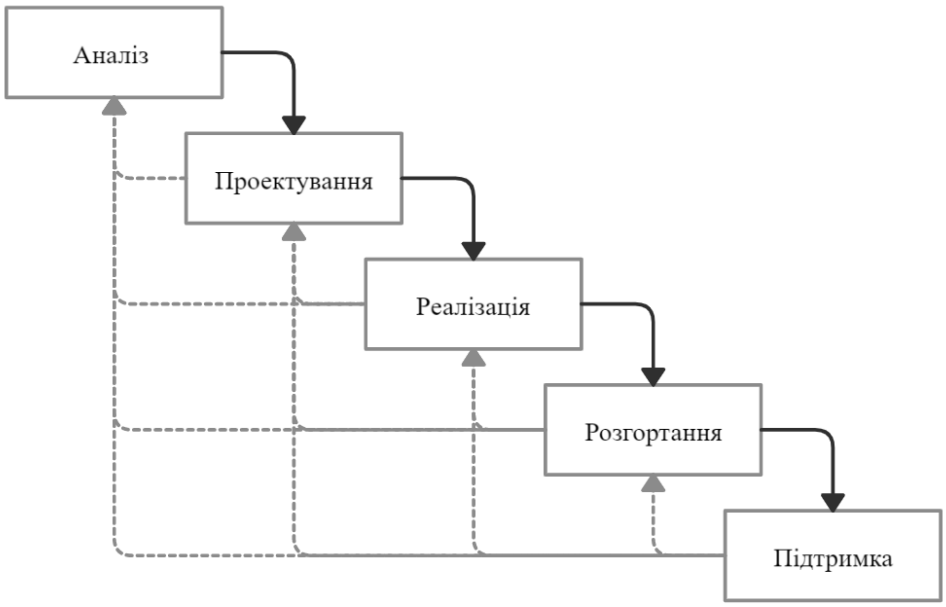


Рисунок 1.1 Етапи розробки програмного забезпечення

Насамперед це етап обддумання можливостей і характеристик програмного забезбечення, що розроблятиметься. Даний аналіз спрямований на визначення як функціональних (функції що виконуватиме система), так і нефункціональних вимог (якість запропонованого рішення). Цей етап також передбачає з'ясування загальної ідеї системи, зацікавлених осіб, яким потрібна нова система, і умов, в яких буде використовуватися система. Виявлені вимоги обробляються з метою створення високорівневої моделі даної системи, що буде абстрагована від непотрібних подробиць проблемної області [3].

Проектування призначене для опису рішення, яке повинно відповідати функціональним вимогам і вимогам ефективності, а також обмеженням того середовища, в якій вона буде працювати. Раніше зібрані вимоги уточнюються і поліпшуються, щоб задовольняти можливим технологічним обмеженням.

1.1. Основні проблеми в проектуванні

Найважливішим етапом створення програмного забезпечення є проектування його архітектури. Для початку необхідно розібрати саме поняття “архітектура програмного забезпечення”. Сам термін значною мірою суб'єктивний і має безліч суперечливих тлумачень, загалом його можна описати як спосіб організації системи, втілений в взаємозвяку її компонентів, а також в принципах, що визначають особливості проектування програмного додатку та його подальшу еволюцію. В процесі проектування системи, зазвичай, приймаються певні рішення, що визначають поведінку системи та не передбачають можливість подальшої зміни цієї поведінки.

В основному організація програмної архітектури повинна вирішувати проблему потенційної складності, яка властива великим корпоративним програмним системам. Зазвичай, складність системи зростає значною мірою швидше ніж її функціонал, тому, досить часто, в проектах із заздалегідь не передбаченою якісною організацією майбутнього коду, виникає момент, коли подальша підтримка цієї складності стає неможливою. обмеженням.

1.2. Особливості вибору архітектури системи

Тому є дуже важливим на стадії проектування вдало вибрати архітектуру системи для мабутнього заощаддження часу та зусиль. Добре спроектована архітектурна модель повинна бути гнучкою в місцях, які передбачають майбутнє розширення, але бути закритою для змін в усіх інших. Також правильний вибір архітектури робить легким подальший супровід, тестування та підтримку системи в майбутньому.

До ключових показників, що характеризують добре побудовану архітектуру належать:

1. *Працездатність та висока ефективність функціоналу. В першу чергу, незважаючи на інші показники, система повинна належно вирішувати функціональні завдання, що стоять перед нею.*

2. Гнучкість системи. Вимоги завжди будуть з'являтися і змінюватися, з розвитком системи. Показником правильно спроектованої архітектури є здатність швидко змінювати систему, не відходячи від першочергово поставленої функції системи, тому завжди необходно, під час аналізу її предметної області та опису моделі, знаходити та оцінювати місця програмного додатку, зміна яких може знадобитися в майбутньому, щоб надалі не витрачати час на зміну підсистем, якщо, така можливість взагалі буде існувати.

3. Можливість розширення системи. Наступним важливим показником є здатність створення нового функціоналу в системі, не змінюючи при цьому її загальної структури та поведінки, за найкортоший можливий проміжок часу.

4. Продуктивність системи. Високонавантаженність та швидкодія є ключовою вимогою до більшості систем. Програмне забезпечення повинно справлятися із навантаження зі сторони користувачів та надавиати відповідь за прийнятний проміжок часу. Із зрстанням навантаження на систему, варіантом розширення є здатність до масштабування.

5. Можливість тестування системи. Належне тестування коду буде не лише зводити кількість помилок до мінімуму, але й буде показником правильного вибору архітектури системи, оскільки це є показником низької зв'язності компонентів системи. Таим чином розробка на основі тестування є окремою методологією по створенню програмного забезпечення на основі тестів.

6. Можливість повторного використання системи. Можливість використання компонентів системи в інших програмних додатках є ще одним показником правильного вибору архітектури.

До показників неправильно сформованої архітектури належать:

1. Жорсткість системи. Такий програмний додаток насилу піддається модифікації, тобто зміна одного компоненту тягне за собою зміну інших частин системи.

2. Крихкість системи. При додаванні нових або зміні існуючих системних елементів, інші частини системи показуть некоректну роботу.

3. Висока зв'язність системи. Неможливо протестувати окрему роботу кожного елемента системи через сильний зв'язок всіх компонентів між собою.

Слід дотримуватися базових концепцій програмного проектування та декомпозиції під час створення інформаційних систем, оскілки це допоможе не тільки на стадії розробки програмного додатку, але й в майбутньому, на стадіях супроводження та розширення системи.

1.3. Огляд популярних архітектурних рішень

На сьогоднішнай день велику популярність отримали два основні архітектурні рішення для створення складних інформаційних систем: багатошарова монолітна архітектура (зазвичай, тришарова) та розподілена сервіс-орієнтована архітектура. Наведені підходи мають свої як переваги так і недоліки.

1.3.1. Монолітна багатошарова архітектура

Монолітна архітектура зручна для роботи невеликих взаємозалежних груп. Використання цього підходу передбачає, що програмне забезпечення буде самодостатнім. Ця архітектура є традиційною для створення веб-додатківми та незважаючи на тривалість існування досі залишається

ідеальним рішенням при виборі архітектури для теперішніх проектів. Основним принципом цього архітектурного підходу є розподіл системи на ключові функціональні частини (рівні). Найпопулярнішим прикладом багатошарового моноліту є тришарова архітектура, що передбачає наявність наступних послідовно зв’язаних рівнів системи: клієнтський рівень, рівень бізнес-логіки та рівень доступу до даних.

Рисунок 1.2 — Схема класичної тришарової архітектури

Розглянемо кожен із трьох рівнів більш детально та опишемо їхні функції.

Перший - клієнтський рівень, який взаємодіє безпосередньо зі користувачем, найчастіше через графічний інтерфейс, та передає вхідні дані для роботи функціоналу системи. Цей рівень не має прямого зв'язку з рівнем доступу до даних та не містить основну бізнес-логіку системи або інформацію про стан програми. На даному рівні, зазвичай, знаходиться найпростіша бізнес-логіка: авторизація користувача, валідація введених користувачем

значень, нескладні операції з даними, що вже завантажені на клієнтський термінал.

Наступним є рівень бізнес-логіки, який містить алгоритми та функції, що задають та описують предметну область. Даний рівень повинен бути найбільш гнучким, оскільки саме він у майбутньому може розширюватись.

Останнім є рівень доступу до даних. Найчастіше це реляційна база даних, доступ до якої виконується відповідними API з рівня бізнес-логіки.

Переваги монолітної архітектури:

1. *Спрощена розробка та розгортання. Є можливість інтегрувати велику кількість різноманітних інструментів для полегшення розробки. Оскільки всі дії виконуються з одним блоком, не виникає зайвих труднощів на стадії розгортання системи.*
2. *Менше наскрізних проблем. Монолітні додатки легко справляються із міжкомпонентними завданнями, такими як ведення контрольних журналів та логів або обмеження швидкості, завдяки своїй єдиній кодової базі.*
3. *Краща продуктивність. При правильній збірці монолітні додатки, зазвичай, забезпечують швидкий зв'язок між програмними компонентами та відповідно показують кращу продуктивність всієї системи.*

Мінуси монолітної архітектури:

1. *З часом кодова база стає громіздкою. Після розгортання більшість продуктів продовжують розширюватись і збільшуватись в обсязі, внаслідок чого їх структура стає розмитою. Кодова база стає громіздкою і в результаті складною для розуміння і змін, особливо для нових розробників.*
2. *Складно інтегрувати нові технології. Необхідність впровадження нової технології в монолітну систему, зазвичай означає*

зміну значної частини програмного коду, що є дорогим і вимагає багато часу.

3. Обмежена гнучкість. У монолітних додатках кожне оновлення вимагає повного повторного розгортання. Ситуація ускладнюється ще сильніше, коли над одним проектом працює декілька команд.

1.3.2. Сервіс-орієнтована архітектура (SOA)

Наступним архітектурним рішенням для створення програмного забезпечення є сервісорієнтована архітектура (SOA). SOA - це стиль архітектури програмного забезпечення, що передбачає систему як сукупність слабосвязанних компонентів, кожен з яких виконує конкретну функцію. Цей підхід розділяє компоненти системи за двома основними ролями: постачальник і споживач сервісів. Обидві з яких можуть виконувати програмні компоненти. Концепція SOA передбачає, що всі модулі системи легко інтегруються і можуть бути легко використані повторно. Натомість компоненти повинні задовільняти наступним потребам [3]:

1. *Єдиний стандарт інтерфейсів для компонентів.*
2. *Абстракція*
3. *Слабка зв'язність компонентів.*
4. *Автономність кожного компоненту*
5. *Можливість використання компонентів в інших системах*

Кожен компонент (сервіс) представляє собою окремий функціональний модуль, який може фізично знаходитися на іншій обчислювальній машині, взаємодія між цими модулями здійснюється через комп'ютерну мережу. В елементарному представленні існує три основні елементи такої системи: постачальник сервісу, реєстр сервісів та споживач сервісу. Алгоритм взаємодії між елементами виглядає таким чином: постачальник сервісу вносить дані про свій сервер у реєстрі, а споживач звертається до реєстру із запитом на

зареєстрований сервіс. Спілкування здійснюється за одним із можливих уніфікованих протоколів передачі даних (SOAP, XML).

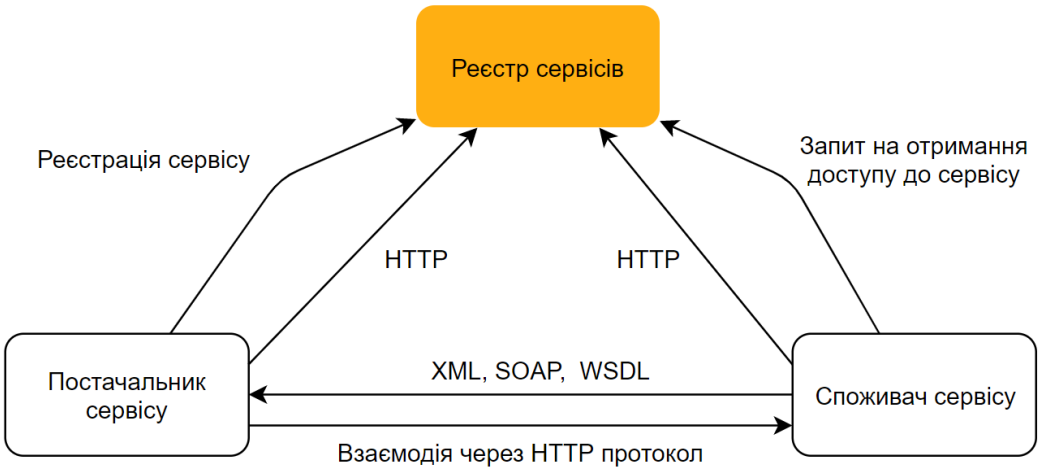


Рисунок 1.3 – Базова схема взаємодії елементів в SOA

Переваги SOA:

1. *Можливість повторного використання сервісів. Завдяки автономності і слабій зв'язності компонентів в сервіс-орієнтованих додатках є можливість повторного використання цих компонентів у інших програмних додатках, без втручання в решту компонентів системи.*

2. Простота в супроводі. Оскільки кожна служба програмного додатку є незалежною одиницею, її легко оновлювати і підтримувати, не вносячи зміни в інші служби. Тому великі корпоративні додатки легше управляються, коли вони розбиті на служби.

3. Висока надійність. Окремі сервіси легше налагоджувати і тестувати, ніж великі звязані модулі, як в монолітах. Завдяки цьому системи на основі SOA більш надійні.

4. Паралельна розробка. Сервіс-орієнтована архітектура підтримує паралелізм в процесі розробки, оскільки вона розбита на

незалежні компоненти. Всі сервіси можуть розроблятися паралельно і бути завершені одночасно.

Недоліки SOA:

1. *Складність в управлінні. Основним недоліком сервіс-орієнтованої архітектури є її складність. Кожен сервіс повинен забезпечувати своєчасну доставку повідомлень. Велика кількість цих повідомлень може суттєво ускладнити управління всіма службами.*
2. *Високі інвестиційні витрати. Розробка системи з використанням сервіс-орієнтованого підходу вимагає значних попередніх інвестицій в людські та технологічні ресурси.*
3. *Додаткове навантаження. Оскільки взаємодіє між сервісами відбувається шляхом обміну повідмлень, використання декількох сервісів може значно збільшити час відгуку і знизити загальну продуктивність системи.*

В підсумку SOA найкраще підходить для складних корпоративних систем. Наприклад банківську систему надзвичайно складно розділити на мікросервіси, а монолітний підхід не підходить, оскільки неполадки в одному модулі приведуть до неправильної роботи всього додатку. Тому найкраще рішення для такої системи - використання SOA – підходу та організація складних модулів системи в окремі незалежні сервіси.

1.3.3. Огляд мікросервісної архітектури

Мікросервіси - це один із типів сервісно-орієнтованої архітектури програмного забезпечення, що продовжує розвиток ідеї розподілу компонентів системи, під час якого система складається із певного набіру невеликих вузьконаправлених сервісів. Основною концепцією MSA є організація ряду автономних компонентів, що незалежно розгортаються, працюють в окремих процесах та взаємодіють один із одним через легкі механізми, як правило HTTP, а разом складають повноцінну систему. Кожен

сервіс будується з урахуванням бізнес-потреб системи та реалізує контректну предметну область із чіткими межами. В той же час кожен компонент може бути реалізований з використанням різного стеку технологій, вибраних у відповідності до функціональних потреб обраної предметної області сервісу.

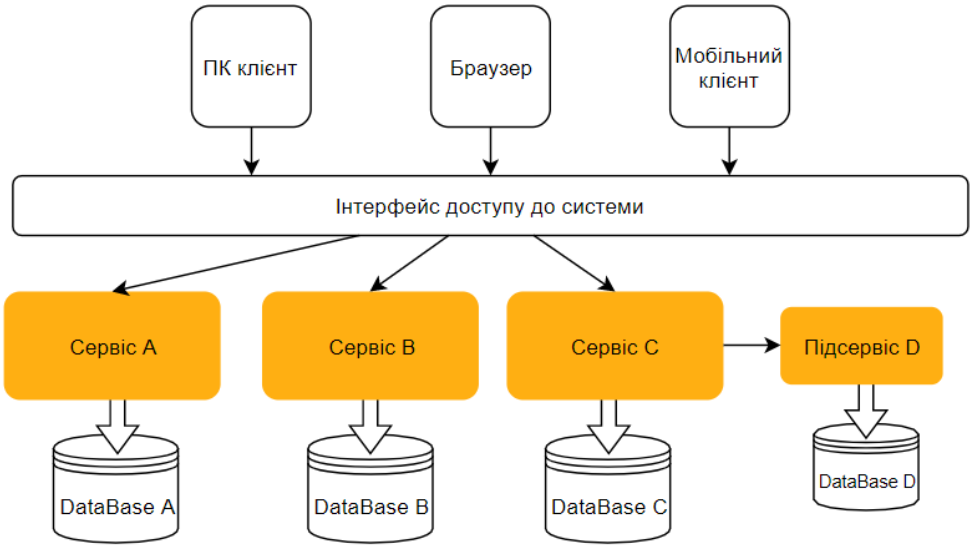


Рисунок 1.3 – Базова схема мікросервісної архітектури

Мікросервісна архітектура розглядається як один із підходів реалізації сервісно-орієнтованої архітектури, оскільки по анології до SOA, вона розподілена та орієнтована на сервіси, що використовують стандартизовані протоколи обміну даними [5].

Стиль розбиття системи на окремі незалежні компоненти не є технічною новизною, він використовує принципи проектування програмного забезпечення, які використовувались ще під час розробки Unix. Ключовим моментом в представленні мікросервісної архітектури є ідея того, що кожен сервіс являє собою невеликий та чітко сфокусований на своїй задачі програмний елемент.

Межі сервісів формуються на основі кордонів визначених бізнес-логікою системи, що дозволяє з легкістю визначити місцезнаходження коду для заданої сфери функцій що виконуються. Утримання сервісу в чітко

визначених межах, не дозволяє його надмірного розростання та запобігає зіткненню із усіма можливими в цьому випадку труднощами. Обмін даними між самими сервісами ведеться через мережеві виклики, це укріпляє відособленість сервісів і допомагає уникнути ризиків, пов'язаних з міцними зв'язками компонентів системи.

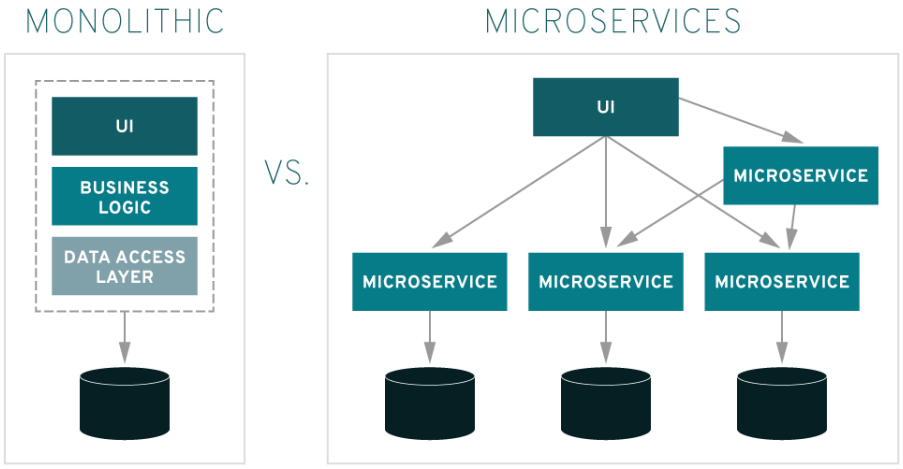


Рисунок 1.4 – Порівняння монолітної та мікросервісної архітектур

Кожен мікросервіс є самостійною програмною одиницею, яка може розгортуватись у вигляді окремого сервісу, або бути компонентом зовсім іншої системи, виконуючи при цьому свої звичайні фунції. Для забезпечення цієї можливості кожен сервіс повинен мати свій програмний інтерфейс - Application Programming Interface (API).

Важливим аспектом кожної системи є організація та управління даними. Мікросервісний підхід, зазвичай, передбачає децентралізоване управління даними. Тобто в той час як для монолітного додатку існує одна база даних, дані із якої використовуються різними компонентами системи, мікросервісна архітектура, робить можливим сценарій при якому всі компоненти оперують власними окремими базами даних. Ключовою перевагою такого підходу є можливість використовувати будь-яку кількість різноманітних технологій збереження даних в одному проекті. Тобто, базуючись на бізнес направленості

конкретного мікросервісу, для нього може бути застосована оптимальна база даних.

Проте мікросервісна архітектура постійно піддається критиці з самого моменту її формування, серед нових проблем, котрі виникають при її впровадженні відзначаються:

- *Мережеві затримки: якщо в модулях, що виконують кілька функцій, взаємодія локально, то мікросервісна архітектура накладає вимогу атомізації модулів і їх взаємодії по мережі;*
- *Формати повідомлень: відсутність стандартизації та необхідність узгодження форматів обміну фактично для кожної пари взаємодіючих мікросервісів призводить як до потенційних помилок, так і складнощів налагодження;*
- *Баланс навантаження і відмовостійкості*
- *Складність операційної підтримки — без хороших DevOps-інженерів складно використовувати мікросервіси, адже необхідна підтримка безперервного розгортання і автоматичного моніторингу.*
- *Тестування мікросервісів може бути громіздким. Використовуючи моноліт, нам потрібно тільки запустити додаток на сервері і переконатися в зв'язку з базою даних. А в мікросервісах, кожен окремий сервіс повинен бути запущеним перед початком тестування [8].*

1.3. Переваги та недоліки мікросервісів

1.3.4. Переваги мікросервісної архітектури

Більшість переваг мікросервісної архітектури притаманні іншим системам із розподіленою архітектурою, проте володіють власними особливостями. Головною метою кожного архітектурного рішеннє є можливість управління складністю системи, і мікросервісний підхід у цьому

абсолютний фаворит. Яким би чином система не розросталась, можливість розподілу на мікросервіси ніколи не дасть їй потонути у своїй складності. Всі переваги та недоліки мікросервісного підходу можна умовно можна на два аспекти: програмний та апаратний [3].

Апаратними перевагами є:

1. Легка масштабованість. Кожен сервіс може бути масштабовано незалежно від всієї системи, зідси випливає можливість виділяти додаткові апаратні ресурси тільки на ті частинисистеми, які цього безпосередньо потребують. Для монолітних систем прийшлось би розширювати всю апаратну базу навіть у випадку коли покращення продуктивності необхідне тільки невеликій її частині.

2. Можливість окремого розгортання. Використання мікросервісів передбачає можливість зміни окремого модуля та дозволяє його розгортання окремим додатком незалежно від системи. Так у разі виникнення проблеми при розгортанні окремого сервісу, він швидко ізолюється та виправляється, незалежно від інших сервісів та системи вцілому.

3. Гетерогенність технологій. Для кожного мікросервісу існує свобода вибору технологій на яких він будуватиметься, без прив'язки. Кожен модуль може використовувати той стек технологій, який надає найбільш оптимальний набір інструментів для вирішення поставленої йому задачі, це стосується не тільки кодової бази модуля, але й сховища даних яке він буде використовувати.

4. Висока стійкість системи. При некоректній роботі системи, помилка буде швидко локалізована, а несправний сервіс переданий на виправлення. В цей час система не потребує зупинки, а відсутній модуль можливо замінити на модуль із таким же функціоналом.

Зі сторони програмних переваг виділяються:

1. Модульність. Мікросервісна архітектура дозволяє зберігати модульність та інкапсуляцію кожного елемента системи, це забезпечує логічне розбиття програмного додатку на модулі, за рахунок наочного фізичного поділу по окремим серверам. В свою чергу така фізична ізольованість гарантує захист від порушення меж між окремими контекстами у системв.

2. Також є очевидним, що мікросервіси легші для розуміння та підтримки, тобто, при роботі над окремим компонентом, програмісту не є необхідним розуміння усіх подробиць роботи всієї системи. Крім того, вибір апаратного забезпечення та інструментарію євільним для кожного сервісу такої системи.

3. Значно менше часу займає запуск та розгортання системи на базі мікросервісної архітектури, ніж ці самі дії для класичного монолітного тришарового додатку.

4. MSA надає можливість безперервного розгортання. Завдяки цьому підходу кожен сервіс масштабується незалежно від всіх інших. Тобто є можливість розгорнути необхідну для задоволення бізнес-потреб системи кількість паралельних екземплярів одного сервісу.

Більшість переваг мікросервісної архітектури властиві будь-яким іншим розподіленим системам, проте саме в цьому підході вони отримали свою ідеальну реалізаціюНедоліки мікросервісної архітектури

1.4.2. Недоліки мікросервісної архітектури

Мікросервісна архітектура не створена для вирішення всіх можливих проблем у проектуванні систем, тому має і ряд значних недоліків. Насамперед виділяється факт ускладнення системи, що не допомагає роботі розробників. Система стає складнішою для розгортання. У виробництві існує також оперативна складність розгортання та управління системою, що складається з

багатьох різних типів послуг. Під час побудови архітектури мікросервісів, виникає багато різнопланових проблем, яких, зазвичай, не очікують під час розробки [8].

Як і створення будь-якої високопродуктивної розподіленої системи, побудова мікросервісного веб-додатку є непростою задачею, звідси з'являються проблеми у впровадженні взаємодії між окремими процесами через обмін повідомленнями або виклик спільного методу через RPC, окрім цього також потребують вирішення безліч питань, пов'язаних із обробкою збоїв у мережі та помилок зі сторони клієнтів.

Ось перелік основних складнощів, що виникають при створенні мікросервісного додатку:

1. *Складність розробки. Оскільки в основі стоїть розподілена система, опрацювання та маршрутизація міжсервісних запитів є завданням що потребує особливої уваги.*
2. *Управління даними. Складнощі у організації розподілених транзакцій, оскільки кожен сервіс може володіти власною базою даних.*
3. *Використання апаратних ресурсів у великих об'ємах. Мікросервісна архітектура вимагає велику кількасть ресурсів, у порівнянні із монолітом, оскільки кожен мікросервіс потребує окремий контейнер із власним програмним середовищем.*
4. *Збільшення навантаження на мережу. Для взаємодії мікросервіси використовують стандартизовані протоколи обміну даними по мережі [4].*
5. *Моніторинг системи. Значно вища складність та вартість моніторингу, проте існують ефективні інструменти для вирішення цієї проблеми.*

ВИСНОВКИ ДО РОЗДІЛУ 1

Задачі проектування можуть бути вирішені великою кількістю ефективних підходів, тому особливо важливо ретельно підійти до вибору архітектури системи, оскільки цей вибір визначає подальші перспективи та напрямок розвитку всього проекту. Побудова складних нетривіальних програмних додатків є об’ємним завданням, тому для його вирішення можливо застосовувати як монолітний, так і розподілений підхід, зважаючи на контекст та бізнес-потреби системи.

Вибір монолітної архітектури є доцільним тільки у випадку створення невеликих та відносно легких систем. Застосування цього підходу для створення об’ємних систем, в результаті підсумку призведе до появи багатьох проблем, що не матимуть вирішення в рамках монолітної архітектури.

Незважаючи на чималу кількість недоліків, на сьогоднішній день мікросервісний підхід є абсолютним фаворитом для основи складних додатків, що планують надалі активно розвиватись та розширювати свою функціональну базу. Наступні розділи дадуть детальну характеристику більшості аспектів мікросервісного архітектурного підходу.

РОЗДІЛ 2

ПОБУДОВА ВЕБ-ДОДАТКІВ НА МСА

2.1. Розбиття веб-додатку на мікросервіси

Більшість прикладів вдалого застосування мікросервісного підходу до архітектури програмних систем, розпочиналися з моноліту, який поступово розростався та ділився. Та в свою чергу не поодинокі випадки, коли проект створювався відразу, як сукупність мікросервісів, але не отримував успіху. З цього випливає, що краще спершу створювати монолітний проект, навіть у випадках, коли система повинна бути досить громіздкою, а вже пізніше, зважаючи на успішність створеного продукту, поступово розділяти її на окремі компоненти. MSA показує себе ефективним та корисним архітектурним підходом, але більшість її переваг матимуть вирішальне значення лише для великих або складних систем. Прості системи ж, значно розумніше буде будувати на основі монолітної архітектури.

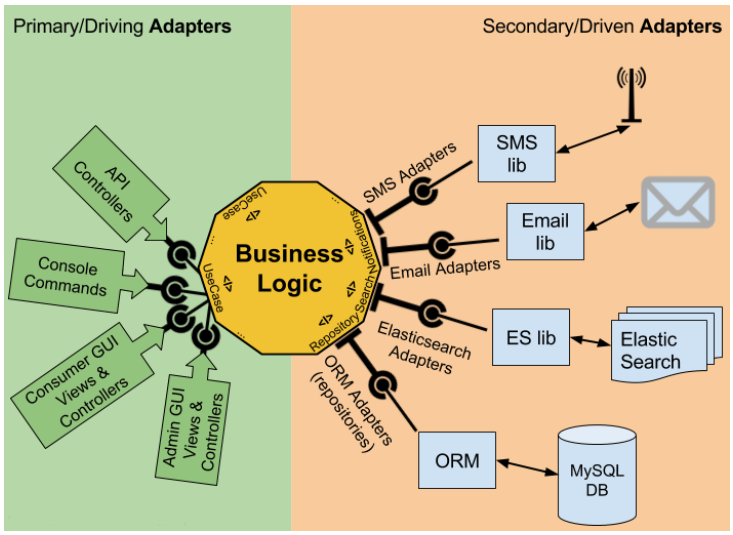


Рисунок 2.1 – Приклад поділу контестів бізнес-логіки монолітного додатку для створення мікросервісів

Дотримання принципу YAGNI (You Aren't Going to Need It), тобто «Вам це не знадобиться», є актуальним у цьому питанні, оскільки часто

використання мікросервісного підходу не є необхідним, а лише несе за собою ряд складнощів та функціональність системи, в якій немає безпосередньої потреби. При розробці нового програмного продукту, спершу, необхідно переконатися в його корисності та доцільності для потенційних користувачів. Найкращим способом перевірити доцільність майбутнього додатку — це розробка його початкової демонстраційної версії. В цьому підході на першому місці знаходиться швидкість розробки (а отже, швидкість отримання відгуків можливих користувачів), в той час коли розробка окремих мікросервісів потребує значної кількості відпрацьованого часу. Іще однією проблемою є факт того, що мікросервісна система працює належним чином лише при досягненні чіткого розділення між окремими сервісами, тобто необхідно створити коректний набір незалежних контекстів.

Будь-яка зміна або допрацювання функціональності між сервісами виявляється складнішою за аналогічний процес у монолітній системі. Під час побудови монолітного додатку значно легше визначити правильні межі для майбутнього поділу системи на мікросервіси. Також це дає додатковий час для підготовки всіх необхідних ресурсів для створення окремих сервісів з чіткими розділеними контекстами. На сьогодні стратегія «спочатку – моноліт» отримала велику кількість різноманітних реалізацій. Розглянемо деякі із них:

- *Логічний шлях – обережний підхід до проектування монолітної системи, коли увага звертається безпосередньо на модульність всіх компонентів програмного додатку, межі API та методи зберігання робочих даних. У випадку коли всі пункти виконано оптимально та з достатньою оцінкою, то майбутній перехід системи на мікросервісну архітектуру не потребуватиме значних затрат та прийняття глобальних рішень.*
- *Наступний популярний варіант розбиття моноліту – створення початкової системи як сукупність декількох сервісів, що по своєму контексту більші за ті, що очікуються в кінцевому результаті.*

В такому випадку є хороша можливість використання цих великих сервісів для отримання досвіду в роботі над мультисервісним середовищем [7].

- *Загальний підхід – почати розробку моноліту і плавно відділяти від нього вузькоспрямовані компоненти, що перетворюватимуться на мікросервіси. При такому підході велика частина початкової системи залишається центральною в мікросервісній архітектурі, проте більша частина нових розробок та розширень відбуватиметься в окремих сервісах, без великого впливу на початкову монолітну частину системи.*
- *Також має право на існування підхід в якому монолітна система, компоненти якої не є модульними, повністю ділиться на мікросервіси. Цей підхід не найефективніший, проте існує і подекуди використовується корпораціями за неможливості вибору іншого, при цьому зі значними ресурсними витратами.*

Незважаючи на чималу кількість переваг підходу «спочатку – моноліт», далеко не всі ставляться до нього однозначно схвально. Контраргументом, насамперед, є те, що починаючи розробку системи з мікросервісів, команда розробників звикає до роботи в такому оточенні та не потребує майбутньої перекваліфікації, як у випадку зміни архітектурного підходу в процесі розвитку веб-додатку як бізнес системи. Також побудова моноліту у вигляді модульних компонентів потребує додаткових зусиль, які можна було б витратити на розробку самостійних мікросервісів.

Мікросервіси дозволяють проводити розробку паралельними командами, відповідальність яких розділяється межами сервісів, це значно пришвидшує розробку, якщо є можливість залучення додаткових робочих ресурсів, тоді як у випадку монолітної системи, це не дало б сильного ефекту.

За основу створення веб-додатку у даній дипломній роботі буде взято підхід до створення початкової системи як сукупності декількох сервісів. І хоча по контексту вони будуть об’ємніші за звичайні мікросервіси, проетування буде відбуватися із можливістю поділу на дрібніші елементи в майбутньому. Створення цих об’ємних сервісів стане хорошою практикою для подальшої роботи в мультисервісному середовищі.

2.2. Інтеграція мікросервісів

Правильна інтеграція є одним із найважливіших технічних аспектів у реалізації мікросервісного архітектурного підходу. У системі, спроектованій належним чином, мікросервіси зможуть зберегти свою автономіюта будуть здатні змінюватися та розвиватися незалежно від решти системи. Проте, помилки, допущені на етапі проектування, в свою чергу, можуть створити ряд складнощів у підтриманні, а деколи навіть поставити під сумнів подальшу працездатність системи [6]. Тому задля уникнення таких, опишемо основні проблеми та підходи до їх вирішення.

2.2.1. Шаблони для проектування мікросервісів

Спочатку розглянемо основні шаблони, що найчастіше використовуються при створенні мікросервісних систем:

1. Шаблон «Агрегатор»

Найпопулярніший шаблон проектування при створенні мікросервісів – «агрегатор». Зазвичай, він являє собою регулярну веб-сторінку, яка регулює виклики сервісів, які, в свою чергу, надають необхідну функціональність веб-додатку.

Оскільки доступ до всіх сервісів відбувається через REST-механізм, у цієї сторінки не виникає проблем в отриманні та обробці даних. Проте, у випадку, коли отримані дані потребують додаткового опрацювання, такого як застосування до них бізнес-логіки, процес ускладнюється та потребує додаткового CDI (Contexts and Dependency Injection)-компонент, що

опрацьовуватиме отримані дані для подальшого успішного відображення. Агрегатор використовується і в тих випадках, коли потрібен мікросервіс вищого рівня, для використання інших сервісів.

2. Шаблон «Посередник»

При роботі з мікросервісами посередник – це один із варіантів паттерна агрегатор, у випадку якого агрегація відбувається на рівні клієнта із можливістю залучення до цього прочесу додаткового мікросервісу. Даний шаблон, як і агрегатор, може масштабуватися незалежно. Це особливо допомагає у випадках, коли кожен окремий сервіс необхідно запускати через власний інтерфейс.

Реалізація цього шаблону, також, може нести формальний характер, в такому варіанті представлення він просто пересилає запит одному із доступних сервісів. Також перед самою відправкою дані можуть перетворюватися в залежності від отримувача. Наприклад, визначення рівня представлення для різних пристроїв може бути інкапсульовано в окремий механізм посередника.

3. Шаблон «Гілка»

Даний шаблон проектування розширює вище згаданий шаблон «Агрегатор» та забезпечує одночасну обробку повідомлень від двох взаємовиключених ланцюгів мікросервісів. Цей патерн, залежності від потреб, також, використовується для виклику різних мікросервісних ланцюгів, або одночасно одного і того ж ланцюга.

3. Шаблон «Дані спільного використання»

До принципів проектування мікросервісів, також, належить автономність. Тобто кожен сервіс є повностековим і контролює всі свої вихідні компоненти: інтерфейс користувача, проміжне програмне забезпечення та транзакції в межах власної бази даних. Тобто сервіс може бути багатомовним і вирішувати поставлені перед ним задачі за допомогою

найбільш доречних інструментів. Наприклад, коли в ході проектування є можливість застосувати NoSQL сховище даних, це буде найкращим варіантом, адже не доведеться підключати громіздку SQL базу даних до одного мікросервісу. Однак, нормалізація бази даних таким чином, щоб кожен мікросервіс володів тільки строго визначеним об'ємом даних, є іще однією великою проблемою при рефакторингу існуючого монолітного додатку.

По принципу цього шаблону проектування, кілька мікросервісів можуть працювати як ланцюг та використовувати спільні бази даних та сховища кешу. Цей шаблон найчастіше буває доречним в особливих бізнес-ситуаціях, коли між двома сервісами виникає сильний зв'язок. Також він часто розглядається як проміжний етап, перед тим як всі мікросервіси стануть незалежними.

5. Шаблон «Асинхронні повідомлення»

Незважаючи на поширеність та зручність, REST – підхід містить важливе обмеження: синхронність взаємодії між елементами розсилки, тому є блокуючим. Впровадження асинхронності в кожному програмному додатку виконується по-різному. Тому, задля подолання цього обмеження, багато мікросервісних систем надають перевагу обробці черги повідомлень, а не класичній REST – моделі (запит/відповідь). Також можлива комбінація обох, наведених вище, підходів задля досягнення особливих цілей систем.

2.2.2. Типи комунікації мікросервісів

В основі наведених вище шаблонів проектування лежать два основні підходи по здійсненню комунікації між мікросервісісами: синхронна (REST та RPC) та асинхронна пересилка повідомлень.

Спершу, на прикладі REST-стилю взаємодії компонентів системи, розглянемо синхронний підхід. Цей стиль має велику кількість базових принципів та обмежень, але в рамках нашої роботи будуть розглянуті лише ті особливості, які мають беспосереднє значення при побудові мікросервісів, а саме пов'язані із інтеграційними складнощами. В рамках REST-стилю

важливим є розуміння ресурсів, якими управляє окремий сервіс, таких як сутність користувача системи. Виконуючи запит сервер маніпулює різними екзкмплярами цього об'єкту. Зазвичай, для упакування та передачі інформації про такі екземпляри, у REST-підході використовується JSON-представлення об'єкта, хоча в базі даних він може зберігатись в абсолютно іншому форматі.

Отримавши екземпляр об’єкту, окремий сервіс може змінювати його характеристики та властивості паралельно зберігаючи ці зміни в базі даних, або передавати на подальше опрацювання інших сервісів.

Також використання HTTP-протоколу значно спрощує розробку з реалізацією REST-підходу. Специфікація HTTP містить в собі ряд особливостей що покривають більшість потреб в розробці комунікації на основі REST. Сам HTTP протокол володіє декількома корисними властивостями, які допомагають при реалізації REST-стилю. В специфікації HTTP протоколу визначаються GET, POST і PUT методи, які уже в своїй назві несуть зрозумілий сенс, який надалі розкриває їх характер роботи із об’єктами взаємодії в системі. Наприклад, GET-метод дістає необхідний ресурс ідемпотентним способом, в той час коли POST-метод ініціалізує створення нового об’єкту в системі.

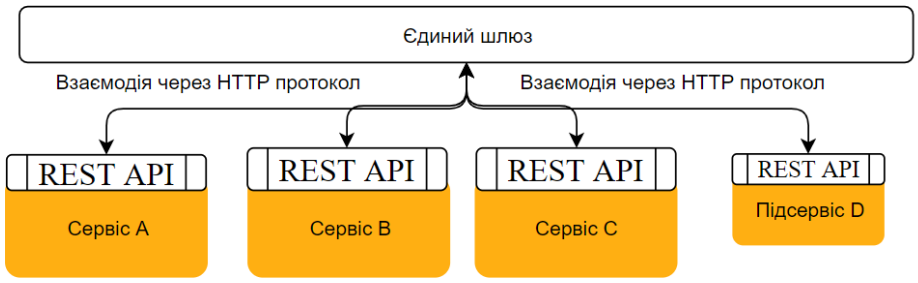


Рисунок 2.2 – Схема комунікації сервісів у REST-підході

Розглянемо асинхронний спосіб обміну даними, тут важливо виділити два основні моменти: створення події у мікросервісах та відловлення початку тієї чи іншої події клієнтом. Серед традиційниго інструментарію, який

охоплює відразу обидві задачі можна виділити RabbitMQ брокер повідомлень із наступним принципом дії.

Компоненти-постачальники використовують API для об’явлення брокеру про початок події. Брокер опрацьовує підписані на нього компоненти-споживачі, надаючи їм інформацію у випадку настання події. Однією із особливостей таких систем-брокерів можна виділити можливість обробки стану споживачів, тобто вони можуть вирішувати, чи сповіщати про настання тієї чи іншої події споживачу, беручи до уваги спсок подій, на які він раніше відповідав.

Необмежена кількість компонентів-постачальників може одночасно відправляти повідомлення в один і той же потік, так само як і необмежена кількість компонентів-споживачів може переглядати повідомлення із одного потоку. В наведеному прикладі споживач – це підпрограма, яка знаходиться в стані очікування повідомлень від постачальників.

Системи що використовують REST-підхід, зазвичай розробляються із підтримкою можливості розширення у майбутньому. Хоча інколи в такому підході виникають ускладнення в процесі розгортання. Але, зрештою, незважаючи на можливі труднощі, REST-підхід є хорошою практикою для реалізації слабозв’язаних архітектур, що керуються на основі подій.

2.3. Розгортання мікросервісів

Важливою характеристикою при оцінці процесу розроблення будь-якого ресурсу є Time to market. Ця характеристика оцінює наскільки швидко розроблена функціональність може бути використана кінцевим користувачем.

При використанні монолітної архітектури, встановлення нової версії навіть одного сервісу вимагає переустановлення всієї програмної системи. Так як все більше і більше ресурсів розгортаються в «хмарі» – постійне переустановлення всієї програми дуже витратне. Крім того, встановлення всього продукту займає немало часу, тому ресурс стає недоступним на час

встановлення, що, в свою чергу, може бути недопустимим для деяких веб-ресурсів.

Мікросервісна архітектура дозволяє перевстановлювати тільки ті сервіси, в яких були зміни. Це дозволяє зменшити час розгортання, а також дозволяє не блокувати інші сервіси в процесі. Крім того, при встановленні лише конкретних сервісів простіше контролювати стабільність ресурсу (при виникненні помилок або несподіваної зміни поведінки простіше виявити джерело) [7].

Для початку розглянемо CI-підхід безперервної інтеграції (Continuous Integration). Використання цього підходу має на меті підтримку загального стану синхронізації системи, яка здійснюється через перевірку належної інтеграції коду кожного нового підключеного сервісу із кодом, що вже існує в системі. Для цього CI-сервер повинен визначати код, який стосується нововведеного сервісу та здійснювати декілька його перевірок, паралельно переконуючись в стані належної компіляції та успішності проходження ним відповідних тестів.

Зазвичай створюється деякий об’єкт-артефакт, частини такого процесу, та використовуються для майбутніх повторних перевірок, у випадках, наприклад, паралельного розгортання існуючого робочого сервісу для маштабування конкретного функціоналу системи. Найчастіше об’єкт-артефакт створюється одноразово та використовується для всіх натупних розгортань тієї версії коду, для якої він був створений. Створені таким чином артефакти поміщаються в окреме сховище, передбачене інструментною базою CI-підходу, для можливого подальшого використання.

Одним із найпрактичніших є підхід, в якому кожн мікросервіс володіє власним сховищем кодової бази, яке використовується тільки для власних CI-збірок. У випадку змін вихідного коду окремого сервісу, відбувається запуск та тестування лише його збірки. Але при такому підході проблеми виникають уже у випадку змін що стосуються декількох сервісів. Для розгортання таких

змін необхідно паралельно запустити процеси збірок та тестування, коду із різних сховиш та забезпечити процес обміну станами між ними.

2.4. Особливості масштабування мікросервісів

Більшість розробників ставлять перед собою пріорітетною умовою створення системи, яка здатна до автоматичного масштабування та адекватної реакції на високі навантаження або збої у деяких її вузлах. Проте для багатьох випадків у проектуванні нових веб-додатків такі вимоги неактуальними та потребують великої кількості затрачених ресурсів.

Зазвичай, коли в рамках проектування системи питання про необхідність масштабування, все-таки, піднімається, насамперед для цього програмного забезпечення висувається додатковий ряд вимог:

- *Доступність функціоналу.*
- *Час відгуку на запити.*
- *Збереження додаткових даних.*

Однією із найважливіших особливостей відмовостійкої системи є можливість безпечно та швидко знижувати об’єм доступної функціональності, без впливу на систему вілому, особливо це важливо, коли функціонал розподіляється між декількома мікросервісами. Повна працездатність системи визначається швидкістю та якістю роботи скомпільованого коду. Проте, у випадку використання мікросервісного підходу, необхідно дивитись на роботу кожного сервісу, як частини системи вцілому. Наприклад, коли один сервіс тісно пов’язаний з декількома іншими сервісами, тим більше показники його працездатності впливатимуть на працездатність інших, зв’язаних з ним сервісів.

Масштабованість описує здатність системи справлятися зі зростаючими навантаженнями. Існує два основних типи масштабування – вертикальне і горизонтальне.

Вертикальне масштабування передбачає збільшення потужності існуючих ресурсів системи. Приклад –встановлення більш потужного процесора на сервер. Горизонтальне масштабування більш складне, тому що найчастіше вимагає зміни в коді системи. Горизонтальне масштабування – процес розбиття системи на більш дрібні частини і розміщення їх на окремих фізичних вузлах. Найпростішим і часто використовуваним варіантом горизонтального масштабування є використання розподілених систем, які дозволяють розподіляти обробку запитів на кілька фізичних вузлів.

Саме в горизонтальному масштабуванні є одна важлива перевага мікросервісів. Так як кожен сервіс є окремою програмою, при підвищенні навантаження на один із сервісів його можна просто встановити на окремий фізичний вузол. Таким чином, завантаження цього сервісу не вплине на стабільність інших сервісів, а також дозволить вертикально масштабувати цей сервіс в залежності від того, які ресурси він споживає найбільше [7].

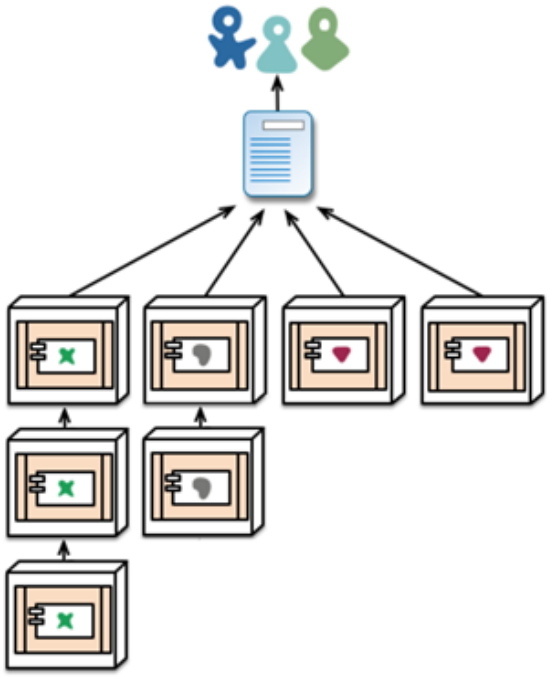


Рисунок 2.3 – Можливість горизонтального та вертикального масштабування мікросервісних систем

Зазвичай, існує дві основні причини для виконання масштабування працездатних систем.

Перша причина – це підвищення відмовостійкості системи: якщо існує ймовірність відмови якого-небуть компоненту системи найкращим способом попередити його буде наявність в системі точно такого ж запасного компоненту.

Друга причина – це підвищення продуктивності, завдяки якому система може витримувати високе навантаження, або зменшити час відповіді, найкраще коли вдається досягнути обох ефектів одночасно.

Якщо для окремих мікрсервісів процес масштабування відбувається досить просто, то масштабування бази даних має ряд своїх нюанчів та обмежень. Для кожного типу баз даних існують свої підходи до масштабування, і разом з цим необхідно розуміти який із підходів буде оптимальним для конкретного випадку використання сховища даних. Незважаючи на те, що більшість сервісів виконує тільки зчитування даних, масштабування цього прцесу виконується значно простіше ніж масштабування запису.

В системах управління базами даних (RDBMS), таких як Oracle або Postgres, існує можливість створення копій схем із основного вузла. В такому випадку один сервіс направлятиме всі запити на запис даних до одного головного вузла, який, в свою чергу, створюватиме з них запити на зчитування даних від декількох реплік.

Через певний проміжок часу після запису даних відбувається генерація резервних копій до реплік з центральної базиданих. Така технологія зчитування даних припускає варіант, коли з’являтиметься розбіжність в актуальності даних, поки не закінчиться процес преплікації. По закінченню цього процесу операцій зчитування зможуть використовувати актуальні дані.

2.5. Інструментна база для використання МСА

Більшість популярних сьогодні мов програмування надають в користування розробникам широкий інструментарій для розробки систем на базі мікросервісної архітектури. Сам по собі мікросервісний підхід передбачає використання технологій, що підтримують можливість використання певних протоколів та інтерфейсів, необхідних для побудови самої архітектури. В свою чергу існує велике різноманіття готових рішень по розробці та розгортанню мікросервісних додатків на сучасних мовах програмування. У цій частині роботи буде розказано про такі рішення на пріоритетних для веб-розробки платформах та особливостях їх використання.

2.5.1. Інструменти для Java

Однією із найпотужніших та найпріорітетніших мов програмування на сьогоднішній день залишається Java. Написані на Java програми представляють із себе скомпільований байт-код, що виконується всередині віртуальної машини Java(JVM), програмою, створеною під відповідну апаратуру, що приймає на вхід байтовий код та по принципу інтерпритатора перетворює його на інструкції для обладнання обчислювальної машини. Однозначною перевагою даного підходу є можливість використання написаного одного разу та скомпільованого коду для запуску на будь-якому пристрої, за умови існування створеноъ для нього JVM.

На сьогоднышный день Java безумовно є найпопулярнішою мовою для створення мікросервісних додатків. Чимало великих компаній вибирають саме її для розширення своїх систем, та створення мікросервісів, найвідомішими прикладами є Amazon та Netflix [4].

Найрозвинутішим та найпотужнішим Java-фреймворком сьогодні є Spring, він надає можливість створення програмних систем як високої так і низької складності. Сам по собі даний фреймворк виник як альтернатива

існуючій J2EE платформі, специфікація якої направлена на розробку веб-систем [10].

Наступним розглянемо Spark framework, легку Java-бібліотека для швидкої розробки веб-додатків. Основною метою при розробці даного фреймворку було забезпечення майбутніх розробників необхідними інструментами для побудови простих та крім того ефективних веб-додатків на мові Java. В своїй основі Spark передбачає спрощення розробки через широке використання лямбда-виразів, що є характерною особливістю Java 8.

Наступним популярним інструментом для Java є Restlet. Це бібліотека, що надає розробникам можливість створення потужних веб-додатків, внутрішні компоненти яких взаємодіють по REST-принципу, тобто через передачу станів об’єктів. Restlet володіє широким набором різноманітних інструментів для веб-розробки та окрім цього доступна для використання на більшості JVM-платформ, таких як Java-EE, GWT або Android.

Та впершу чергу Restlet славиться можливістю легкої конфігурації серверу через параметри налаштувань та дозволяє підключення додаткових функцій, таких як перегляд стану працездатності запущеного серверу.

2.5.2. Інструменти для Python

Розглянемо Python, мову, яка за декілька останніх років стала однією із найпопулярніших у веб-розробці та володіє чималим інструментарієм, який цьому сприяє. Із самої концепції Python створювався як мова, що максимально спиятиме продуктивності розробника і збереженню чистоти та читабельності коду. Python володіє мінімалістичним синтаксисом, проте його стандартна бібліотека містить в собі значний об’єм корисного функціоналу.

Серед основних напрямків використання Python виділяються:

- *Створення веб-додатків*
- *Алгоритми машинного навчання*
- *Аналіз великих об’ємів даних*

- *Швидке створення прототипів бізнес-ідей*
- *Написання проміжних скриптів автоматизації*

За весь час свого існування Python отрмив чималу кількість фреймворків для розробки мережевих систем. Більшість з яких мають на меті не тільки надання комфортної платформи, але й володіють власним унікальним функціоналом, що допомагає у створенні та підтримці веб-додатків. Знайти необхідний фреймворк можливо після првильного аналізу предметної області, для реалізації якої він буде використаний.

Django - це, найпопулярніша веб-платформа Python, це система високого рівня, створена для задоволення найпоширеніших вимог, які виносяться для веб-додатків.

Архітектура Django схожа на шаблон проектування MVC, тому Django добре підходить для веб-додатків, що взаємодіють із базами даних. Також він містить власну реалізацію ORM та може автоматично генерувати моделі на основі схем у базі. Після визначення моделей системи, можливо застосувати розширений API Python для їх використання.

Якщо для конкретної задачі по розробці веб-додатку Django виявляється занадто громіздким, варто звернути увагу на Flask, значно легший фреймворк для розробки мережевих сервісів на мові Python. Серед інших його особливостей можна виділити існування додаткової бібліотеки Flask-Injector, як видно із назви вона пропонує розробку слабкозв’язаних систем на основі шаблону DI (впровадження залежностей) [13].

Сам фреймворк Flask базується на двох великих зовнішніх бібліотеках: шаблонізаторі Jinja2 та інструментній-бібліотеці для реалізації WSGI-стандарту взаємодії Wekzeug. Зазвичай для розробки мікросервісів на мові Python обирають цей фреймворк, оскільки саме він надає широкі можливості для реалізації системи на основі слбко пов’язаних компонентів.

Також має право на розгляд асинхронна мережева бібліотека Tornado. Завдяки використанню неблокуючих мережевих операцій вводу-виводу, Tornado дозволяє підтримувати величезну кількість одночасних відкритих з'єднань. Популярне використання цього фреймворку для розробки систем веб-сокетів та мережевих додатків, що потребують довгих безперебійних з'єднань одночасно для великої кількості користувачів [14].

Іще однією бібліотекою для створення мікросервісів серед всього різноманіття веб-бібліотек для Python варто виділити Nameko. Вона дозволяє розробникам не зациклюватися на деталях реалізації сервісу, а більше сконцентруватися на бізнес-логіці додатку, надаючи, в свою чергу, широкий інструментарій для тестування.

Значною перевагою Nameko є також можливість зручного масштабування одного сервісу на цілий кластер, що складатиме множину екземплярів одного сервісу. Для цього бібліотека надає розробникам багатий функціонал для реалізації взаємодії, написаних на Python, сервісів з готовим кластером Nameko [15].

2.5.3. Інструменти для платформи .NET

Серед переліку платформ для веб-розробки варто згадати популярну .NET, яка сьогодні широко використовується при розробці високопродуктивних мережевих систем. Платформа володіє багатою стандартною бібліотекою та містить безліч корисних інструментів, для полегшення веб-розробки. Окрім цього основа платформи, мова C#, володіє якостями високорівневих та низькорівневих мов програмування, що дозволяє їй бути зрозумілою на етапі написання коду та ефективною на етапі виконання.

Виділимо головні переваги якими володіє платформа з точки зору створення мікросервісних систем:

1. *Висока обчислювальна продуктивність. За рахунок доступу до апаратних ресурсів, C# значно кращий у швидкодії, в порівнянні з іншими мовами.*

2. *Підтримка багатьох методологій програмування, таких як об'єктно-орієнтоване, аспектно-орієнтоване, узагальнене, функціональне, структурне та інші.*

3. *Доступна велика кількість навчальної літератури по цій платформі, що дозволяє легко її освоїти та швидко розпочати розробку.*

4. *Широкі функціональні можливості даної платформи.*

Розглянемо детальніше фреймворки Nancy, що на мою думку найкраще підходить для створення мікросервісів на цій платформі.

Nancy – це фреймворк з відкритим вихідним кодом, основною ціллю якого є надати розробникам великий та зручний функціонал для розробки веб-додатків та сервісів [16]. Серед особливостей цього фреймворку можна виділити:

- *Простоту роботи. По замовчуванню в ньому заздалегіть задано розумні значення налаштувань та інших формальностей, тому починати розробку на цьому фреймворку просто навіть тим, хто раніше не був з ним знайомим.*

- *Легка адаптація до конкретної задачі. Навіть у випадку коли звичайних налаштувань не достатньо для робзробки частини фонкціоналу веб-додатку, абсолютно все у фреймворку Nancy розбито на компоненти та може бути змамінено на потрібну реалізацію.*

- *Легкість тестування. Оскільки під час проектування цього фреймворку однією із задач була можливість зручного тестування, розробникам надається зручний функціонал для проведення тестів у вигляді окремої бібліотеки Nancy.Testing.*

ВИСНОВКИ ДО РОЗДІЛУ 2

В даному розділі було проаналізовано основні аспекти по розробці мікросервісних систем, такі як поділ системи на незалежні мікросервіси та типи комунікації. Проведено огляд популярних шаблонів проектування, для спрощення реалізації даного архітектурного підходу. Розглянуто додаткові особливості інтеграції мікросервісів, а саме розгортання та масштабування.

Також проведено роботу по огляду доступних рішень реалізації мікросервісів на мовах програмування Java і Python, та платформі .NET. Огляд відповідних бібліотек та фреймворків дозволяє зробити правильний вибір стеку технологій для розробки кожного конкретного мікросервісу.

РОЗДІЛ 3

РОЗРОБКА СИСТЕМИ НА БАЗІ МСА

В даному розділі буде описано процес створення системи з мікросервісною архітектурою на прикладі веб-додатку. Розроблена система надаватиме можливість реєстрації в системі нового користувача, дозволить йому перегляд доступних чат-кімнат, дозволить долучитись до наявних, чи створити власну чат-кімнату, а також надасть функціонал для адміністрування створеного чату. В ціломц, система складатиметься із трьох частин: центрального інтерфейсу користувача, набору функціональних сервісів та спільних механізмів взаємодії.

Далі буде проведено детальніший огляд технологій, вибраних для реалізацій мікросервісного архітектурного стилю в рамках даної роботи, а також описано принципи побудови кожного компоненту системи.

3.1. Огляд вибраного стеку технологій

Спершу, перед визначенням оптимального стеку технологій для створення мікросервісів, виділимо ключові вимоги до системи, що буде розроблюватись:

- *Гарантія слабкої зв'язності компонентів*
- *Наявність механізму швидкої взаємодії між компонентами системи*
- *Управління розподіленими транзакціями*

Розглянемо технології, що найкраще підійдуть для забезпечення кожної із вимог. Для гарантії контекстної незалежності та слабкої зв'язності мікросервісів найкраще підійде мова із можливістю строго окреслювати межі впливу кожного описаною нею компоненту системи. Для організації зручного та швидкого механізму взаємодії сервісів виберемо фреймворк, який надає багатий функціонал для використання відомих протоколів передачі даних та

дозволяє створити REST API точки доступу для кожного сервісу. Забезпечити інтеграцію сервісів допоможуть інструменти, які надаються у вільний доступ компанією Netflix, а саме: Zuul, Hystrix та Eureka. Як сховище даних підійде одна з наявних RDBMS-систем, що дозволить організувати управління розподіленими транзакціями на рівнях декількох сервісів.

Окрім вище згаданих технологій, також, необхідним є сервіс, що дозволяє організувати незалежне розгортання кожного сервісу. Для цього найкраще підійде Docker-контейнер із можливістю попереднього налаштування середовища.

3.1.1. Огляд вибраних мов програмування

На сьогодні мова програмування Java безумовно є найпопулярнішим вибором для створення веб-додатків. За час свого існування в індустрії вона отримала велику кількість розвинених бібліотек та фреймворків для вирішення всеможливих проблем, що виникають у веб-розробці. Тому було вирішено обрати Java провідною мовою у розробці більшої частини нашої системи. Також вагомою перевагою цієї мови перед конкурентами є наявність для неї потужного фреймворку Spring, що надає масу технологічних рішень, на популярні проблеми у розробці веб-додатку.

Серед альтернатив, також, можна звернути увагу на Python. Ця мова показує себе безумовним фаворитом, коли ключовим параметром вибору стоїть швидкість розробки. Його можна помістити до нашого стеку для використання у випадку необхідності швидкого створення невеликих окремих сервісів, або написання скриптів автоматизації роботи середовища.

3.1.2. Огляд вибраних фреймворків

Найкращим інструментом для побудови мікросервісів на Java, безумовно, є фреймворк Spring. Він надає розробникам такі важливі модулі як Spring Boot та Spring Cloud, з якими розробка складних розподілених систем виглядає значно простішою. Spring Boot, як важлива складова інструментного

набору бібліотеки Spring дозволяє розробку простих у реалізації та підтримці Spring-додатків, оскільки налаштування його функціоналу потребує невеликої кількості конфігурацій, у порівнянні з відомою бібліотекою Spring MVC [12].

Серед найбільш значущих особливостей Spring Boot можна виділити:

- *Вбудований контейнер сервлетів Tomcat або Jetty;*
- *Можливість автоматичної конфігурації;*
- *Дозволяє проводити моніторинг стану роботи системи;*
- *Конфігураційні файли прості в написанні [12].*

У свою чергу Spring Boot став базою для розробки складнішого фреймворку Spring Cloud, який призначений для створення систем із розподіленою архітектурою та володіє більшим набором інструментів для веб-розробки.

Перелік основних переваг фреймворку Spring Cloud у порівнянні з аналогами:

- *Готовий механізм реєстру сервісів та системи маршрутизації міжсервісних зв'язків;*
- *Можливість балансування навантаження системи;*
- *Наявність механізму автоматичного вимикача;*
- *Механізм керування розподіленим обміном повідомлень між декількома сервісами. [17]*

Spring Cloud дозволяє використовувати декларативний підхід до написання коду компонентів та надає масштабоване сховище конфігурацій розподіленої системи. Фреймворк надає можливість використання сервісів контролю версій, таких як Git або Subversion та крім цього може використовувати локальні файли.

Spring Cloud надає розробникам у використання ряд зручних анотації для реалізації декларативного підходу програмування та автоматичної конфігурації програмних ресурсів. Крім цього фреймворк дозволяє

використання спрощеного механізму підключення до автономних сервісів та робить можливим розгортання системи на оточеннях, отриманих на таких хмарних платформах, як AWS Cloud або Heroku. Завдяки можливості власної конфігурації, веб-додтки створені на базі Spring-фреймворку здатні легко підключатись до великих хмарних сервісів. Існує можливість використаня як наявних конекторів підключення так власних, написаних під особливості системи. Підключення таких сервісів не вимагає зміни створеного програмного забезпечення на Spring, а відбувається через додавання в область бачення необхідної jar-бібліотеки [17].

Даний фреймворк дозволяє створювати ефективні сервіси шляхом написання невеликих об’ємів коду, тому стане чудовим базовим інструментом для створення нашої системи.

3.1.3. Огляд вибраних технологій зберігання даних

MySQL – це розроблена Oracle система реляційного управління базою даних з відкритим вихідним кодом, основана на мові структурованих запитів (SQL). Робота MySQL підтримується практично на всіх доступних сьогодні платформах, серед яких UNIX-системи та Windows. Найчастіше MySQL використовується для збереження даних веб-додатків або інтернет-публікацій, хоча сама система спроектована для використання в ширшому діапазоні можливостей.

Сьогодні MySQL є однією із найпопулярніших RDBMS-систем, що стоїть за багатьма відомими мережевими системами-гігантами у світі, такими як Facebook, Twitter та YouTube та крім цього використовується незліченною множиною корпоративних та споживчих веб-додатків [18]. MySQL створена на основі клієнт-серверної архітектурної моделі. Ядром системи є MySQL сервер, який виконує обробку всіх інструкцій що надходять до бази даних. Сам MySQL-сервер може бути також використаний як окрема програма в клієнт-серверному веб-середовищі.

Завдяки MySQL стає можливим збереження даних у декількох вузлах-сховищах, а також масштабування таблиць із даними для кращої продуктивності програмного забезпечення, яке використовує ці дані. MySQL має низький поріг входження, адже перед початком роботи користувачам потрібно володіти тільки набором стандартних SQL-команд.

MySQL ідеально підходить для використання у нашій системі саме завдяки своїй простоті. Ця RDBMS-система надає зручний інтерфейс для управління сховищем даних та дозволяє створення складніших розподілених транзакцій на вищих рівнях.

3.2. Огляд інструментної бази

Окрім ядра та основних сервісів, необхідно забезпечити систему додатковими функціональними мікросервісами, які нададуть єдиний програмний вхід до системи, допоможуть організувати вхаємозвязок компонентів (API Gateway) та створять складуть зручний інтерфейс моніторингу стану системи та її регуляції вцілому (Service Discovery, Circuit Breaker).

3.2.1. Інфраструктурні сервіси

Netflix Zuul – це реалізація шаблону проектування API Gateway, надана в вільне використання розробниками від Netflix [9]. Налаштування сервісу Zuul для системи виконується завдяки можливості використання однієї SpringCloud-анотації над звичайним java-класом.

Zuul працює як зовнішній інтерфейс, що створює єдиний вхід до всієї системи та розподіляє запити по функціональним сервісам. В свою чергу кожен мікросервіс в системі володіє власним REST API, через який і здійснюється маршрутизація. Для організації загалної маршрутизації системи створюється конфігураційний файл що описує алгоритм правильного перенаправлення запитів, які сервіс Zuul приймає на вхід.

Для підвищення відмовостійкості системи, веб-додатк застосовуватиме функціонал автоматичного вимикача для кожного сервісу. Інструментом який забезпечить нашу систему таким функціоналом буде сервіс Netflix Hystrix Dashboard.

Принцип робот цього сервісу наступний: коли один із компонентів системи не відповідатиме у зв'язку із виникненням локальних збоїв чи помилок, механізм сервісу Hystrix перенаправлятиме запит на внутрішній метод, створений спеціально для опрацювання таких ситуацій. Якщо збої сервісу повторюються або не припиняються, Hystrix переправлятиме кожен виклики на резервний метод, паралельно моніторячи стан проблемного сервісу. Як тільки сервіс Hystrix виявить що компонент, у якому виникали збої, знову працездатний, то перестане перенаправляти запити на резервний метод. Для визначення статусу працездатності сервісу, на протязі деякого часу Hystrix надсилає йому маркерні запити.

Підключення цього сервісу не відрізняється від попереднього та здійснюється простим функціоналом SpringCloud-анотацій. Метод, який опрацьовуватиме ситуації збоїв у мікросервісах створюється через запис над ним анотації *@HystrixCommand*. Після правильного запуску Hystrix надає зручний функціонал моніторингу станів працездатності кожного компоненту системи у вигляді панелі.

Наступним ключовим моментом у використанні мікросервісної архітектури є завдання виявлення необхідних мікросервісів. Ця задача вирішується через використання сервісу Netflix Eureka, який надає функціонал для реєстрування всіх компонентів-сервісув у системі. Функціонал сервісу можна додати до нашої системи через простий запис анотації *@EnableEurekaServer* над класом Spring Boot додатку.

Після налаштувань та запуску сервісу розробник отримує функціонал у вигляді зручної панелі управління. На цій панелі можна знайти всі зареєстровані в системі сервіси. Також існує можливість доступу до будь-

якого із відкритих сервісів. Якщо REST API сервісу, до якого здійснюється доступ, налаштовано по принципу HATEOAS (Hypermedia as the Engine of Application State), Eureka-сервіс автоматично виявить всі його функціональні можливості.

3.2.2. Система Docker

Оскільки система може складатися із величезної кількості різних по структурі мікросервісів, перед розробниками виникає проблема управління різними середовищами, що виконують запуск цих сервісів. Для інкапсуляції мікросервісів та зручності опису їхніх програмних середовищ прийнято використовувати так звані «контейнери». Наразі, система автоматичного розгортання та контейнеризації Docker є найкращим рішенням для отримання цього функціоналу.

Як інструменту для здійснення контейнеризації компонентів та систем в цілому, Docker можна порівняти з механізмом віртуальних машин, головною метою яких є оптимізації роботи обчислювальних ресурсів системи. Тобто використовуючи декілька віртуальних машин на одному сервері стає можливим розгортання декількох зовсім незалежних систем на кожній машині. При дотриманні цієї моделі, стає можливим створення стабільного програмного середовища для декількох незалежних додатків на одній машині. Проте такий підхід має і ряд недоліків, серед яких неможливість масштабування додатків, через ризик виникнення проблем продуктивності, які у свою чергу породжуються надмірним споживанням апаратних ресурсів віртуальними машинами.

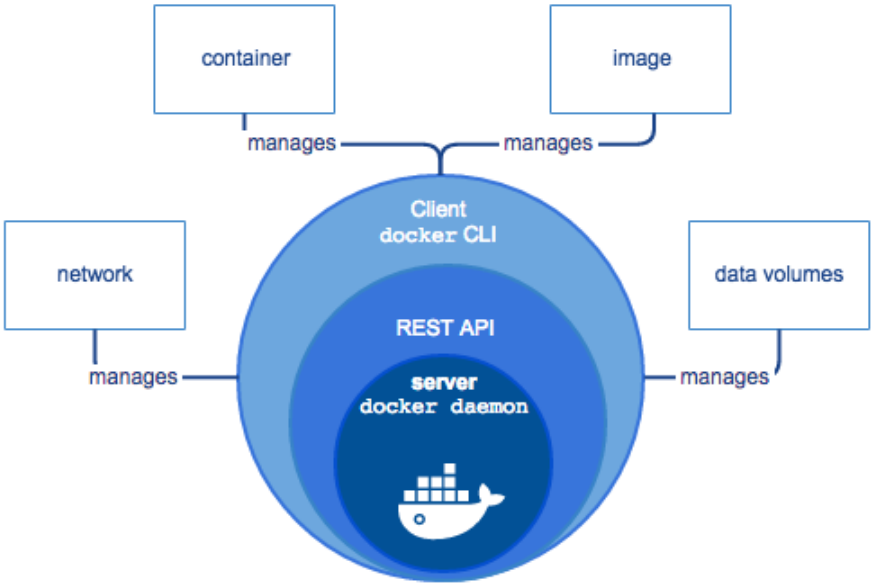


Рисунок 3.2 – Схема концепції системи Docker

В противагу, завдяки своїй легкості та використанні мізерної кількості обчислювальних ресурсів операційної системи, Docker дозволяє уникнути проблем продуктивності при розгортанні великої кількості сервісів в окремих контейнерах на одному сервері.

Для наочності переглянемо ряд переваг системи Docker:

- *Швидке розгортання. Контейнер не потребує налаштування середовища, для його розгортання необхідно тільки створення образу сервісу, який він повинен містити.*
- *Швидкий запуск. Оскільки контейнер по своїй суті є процесом операційної системи, то його запуск не потребує часу більше, ніж запуск якої-небудь легкої прикладної програми.*
- *Механізм контейнерів простий в управлінні та дозволяє проводити масштабування сервісу що в ньому знаходиться без втрати продуктивності операційної системи.*
- *Легкість та оптимальне використання апаратних ресурсів системи.*

- *Docker підтримує розгортання на переважній більшості із можливих операційних систем [11].*

Завдяки своїм перевагам система автоматичного розгортання та контейнеризації Docker є найкращим рішенням для організації управління мікросервісами на власних середовищах. Також причинами його вибору стали простота у використання та спрощення процесу розгортання окремих мікросервісів.

3.3. Деталі реалізації

Розглянемо детальніше процес розробки та підключення сервісів, які міститимуть головний функціонал системи.

3.3.1. User Service

Даний сервіс розроблено на мові програмування Java та з допомогою фреймворку Spring. Об’єктом, яким оперує даний сервіс є користувач. Взаємодія з рештою сервісів відбувається завдяки REST-технології по HTTP-протоколу. User сервіс взаємодіє із Chat сервісом, тим самим дозволяє користувачеві перегляд списку доступних чат-кімнат. Оскільки сервіс володіє інформацією про користувачів, що знаходяться в системі, він також може надавати спсок користувачів та інформацію по кожному, але такий фенкціонал буде відкритим тільки для користувачів що володіють спеціальними дозволами в системі.

Реалізація контроллера відбувається через створеннння Java-класу із спеціальною анотацією *@RestController*, яку надає Spring-фреймворк. Сховищем даних для User сервісу виступає звичайна реляційна база даних.

3.3.2. Chat Service

Мікросервіс надає користувачам доступ до наявних чат кімнат, та функціонал безпосереднього створення та адміністрування власних. Мікросервіс оперує об'єктами ChatRoom в системі. Створення нового такого

об'єкту відповідає створенню нової користувацької чат-кімнати, а зміна його характеристик несе за собою зміну налаштувань.

Через даний сервіс є можливість отримання всіх існуючих чат кімнат, щоб переглянути її деталі або підключити до неї користувача.

3.3.3. Message Service

Даний сервіс потребує особливої уваги, оскільки саме він зіштовхнеться із основним навантаженням системи. Його функціонал передбачає механізм для реалізації процесу відправки та отримання повідомлень між великою кількістю користувачів в рамках однієї чат-кімнати. Для реалізації даного сервісу прийнято рішення використовувати легкий механізм сокетів, що дозволить опрацьовувати процеси відправки та отримання без надлишкових навантажень на систему.

Сокет, що перекладається з англійської як роз'єм – це назва програмного інтерфейсу що забезпечує обмін даними між незалежними процесами. Самі процеси, що беруть участь у цьому обміні можуть виконуватись як на одній, так і на різних обчислювальних машинах, що пов’язані спільною мережею.

В індустрії розрізняють клієнтські та серверні сокети. Клієнтські сокети виконують роль кінцевих апаратів взаємодії, а серверні, в свою чергу, є комутаторами, що забезпечують з’єднання між клієнтами. Прикладні програми, такі як браузер, або додаток-клієнт на машині користувача можуть використовувати лише клієнтські сокети, проте на стороні сервера використовуються як серверні сокети (як етап сполучення між клієнтами), так і клієнтські сокети (для можливості прямого з’єднання з іншими серверами).

У даній роботі буде використано готову реалізацію цього механізму на мові програмування Java, а саме класи Socket та ServerSocket.

Клас Socket дозволяє створити об'єкт-сокет, вказавши IP-адреса і порт з'єднання:

```
Socket messageSocket = new Socket(ipAddress, serverPort);
```

Створений Socket-об'єкт містить в собі вхідний та вихідний потоки. Вхідний потік дозволяє читати з сокета, вихідний, в свою чергу, дає можливість записувати в сокет:

```
InputStream socketInputStream = socket.getInputStream();
OutputStream socketOutputStream = socket.getOutputStream();
```

Оскільки об’єкти InputStream та OutputStream оперують потоками байтів, для зручності використання їх можливо конвертувати їх в потоки даних. Це дозволить передавати та отримувати дані у вигляді тексту, що і повинен виконувати наш сервіс.

```
DataInputStream messageInput =
                new DataInputStream(socketInputStream);
DataOutputStream messageOutput =
                new DataOutputStream(socketOutputStream);
```

Текстові дані, що передаватимуться об’єкту **messageInput** одним користувачем, проходитимуть обробку на стороні серверу та виводитимуться через об’єкт **messageOutput** в рамках однієї чат-кімнати для всіх її користувачів.

3.3.4. Інтеграція сервісів

Інтеграція наших сервісів відбуватиметься через використання інструментів компанії Netflix. Рисунок 3.1 демонструє повний набір компонентів системи та їх взаємодію. Далі розглянемо детальний зв’язок інструментних та функціональних сервісів в системі.

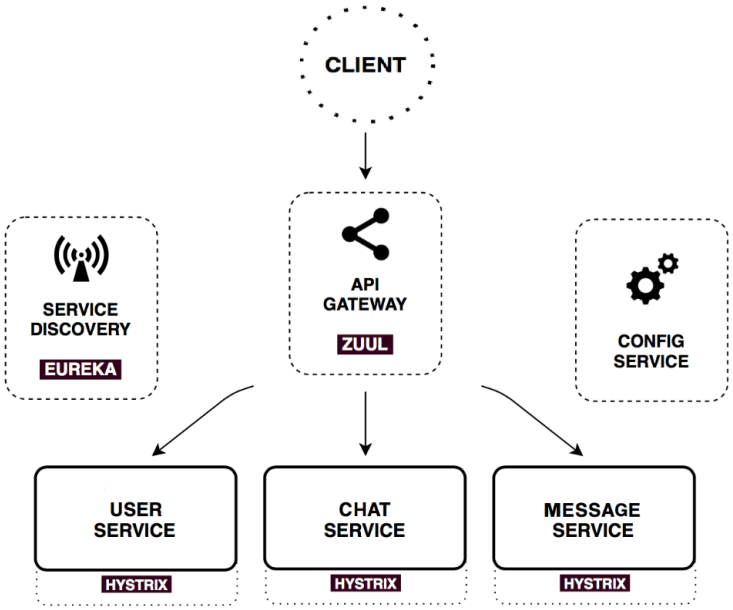


Рисунок 3.1 – Загальна архітектура додатку

Важливим завданням на етапі розробки системи є створення єдиної точки доступу, яка надаватиме можливість взаємодії із усіма компонентами системи. Саме таким інструментом є сервіс Netflix Zuul, особливості якого були описані раніше у цьому розділі. Налаштування даного сервісу в рамках нашої системи створить єдиний інтервейс доступу та дозволить доповнення системи функціональними мікросервісами. Тобто кожен створений нами сервіс, після розгортання, підключатиметься до єдиного шлюзу та надаватиме йому свій API.

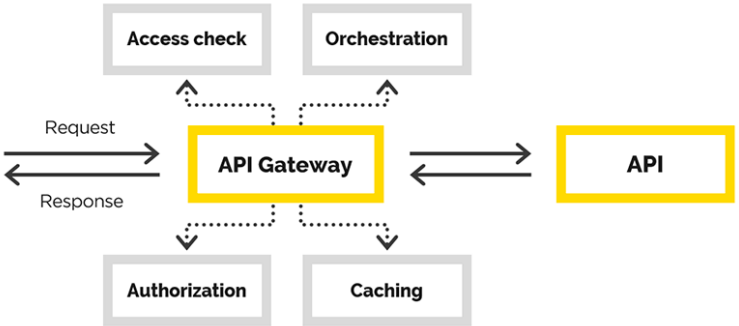


Рисунок 3.2 – Принцип роботи шаблону API Gateway в рамках мікросервісної системи

Наступним важливим елементом системи є автоматичний вимикач (Circuit Breaker), використовувати який ми будемо через реалізацію Netflix Hystrix. Даний сервіс забезпесить додаткову надійність та вібмовостійкість нашої системи. Виклик кожного сервісу попередньо обгортається викликом об’єкту автоматичного. Особливу цінність даний інструмент має для мікросервісних систем, оскільки гарантує необхідну стійкість.

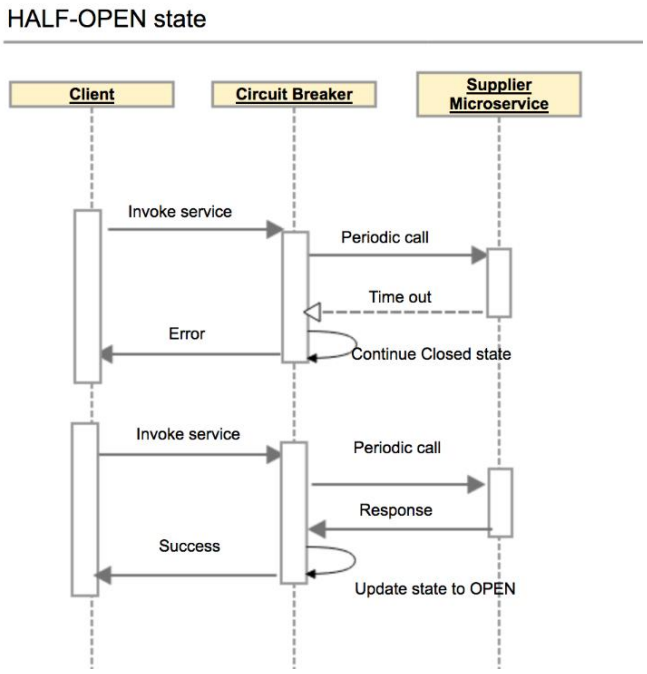


Рисунок 3.3 – Принцип роботи автоматичного вимикача в рамках мікросервісної системи

Задача моніторингу станів кожног сервісу виконується через сервіс Netflix Eureka. Даний сервіс надає можливість реєстрування всіх наших функціональних сервісів у системі. Функціонал сервісу отримується через простий запис анотації *@EnableEurekaServer* над класом Spring Boot додатку.

Кожен створений нами сервіс несе в собі долю функціоналу сукупність якого створює повноцінну систему. Завдяки User Service відбувається управління даними користувачів. Chat Service надає функціонал для доступу, створення та налаштування чат-кімнат. Message Service містить в собі механізм пересилки повідомлень між користувачами. Хоча формально всі

сервери є незалежними, в рамках системи вони працюють як єдиний механізм. Наприклад, відображенні списку доступних для користувача чат-кімнат потребує виклику Chat Service, а для здійснення спілкування в чат-кімнаті використовується Message Service.

Взаємодія між сервісами виконується синхронно через єдиний інтерфейс доступу, що в свою чергу, звертається до REST API відповідного сервісу. Розгортання сервісів відбувається в окремих Docker-контейнерах із попередньо налаштованим для потреб конкретного сервісу середовищем.

3.4. Опис функціональних можливостей системи

У системі сервіси User та Chat є самостійними та складаються з компонентів: «*Model*», «*Controller*» та «*View»*, проте сервіс Message по своїй суті є підсервісом сервісу Chat, тому складається тільки з механізму пересилки повідомлень та бази, яка зберігає об’єкти «*MessageModel*».

Маніпуляції що виконуються користувачем передаються на сервіс Netflix Zuul (реалізацію шаблону API Gateway) та вже з нього запити розподіляються на контролери відповідних функціональних сервісів.

Детальну UML діаграму класів кожного сервісу системи наведено у додатку.

Функціонал сервісу User:

- *UserModel* – клас, що відповідає за створення об’єкту користувача для подальшого використання у системі;
- *UserController* – клас, що містить в собі функціонал демонстрації веб-сторінок, доступних через використання сервісу User;
- *Метод addUser()* – виконує збереження об’єкту UserModel у базі даних сервісу User;
- *Метод loadUserByLogin()* – виконує пошук об’єкту UserModel по вказаному логіна у базі даних сервісу User;

- *Метод editUser()* – виконує збереження змін об’єкту UserModel у базі даних сервісу User;
- *Метод getAllUsers()* – повертає всі об’єкти UserModel що знаходяться у базі даних сервісу User;
- *Метод validRegistrationForm()* – перевіряє форму реєстрації користувача на коректність введених даних.
- *Метод validAuthorizationForm()* – перевіряє форму авторизації користувача на коректність введених даних.
- *Метод hashPass()* – виконує хешування введених під час реєстрації або авторизації паролів

Функціонал сервісу Chat:

- *ChatModel* – клас, що відповідає за створення об’єкту чат-кімнати для подальшого використання у системі;
- *ChatController* – клас, що містить в собі функціонал відображення веб-сторінок, доступних через використання сервісу Chat;
- *Метод addChat()* – виконує збереження об’єкту ChatModel у базі даних сервісу Chat після створення користувачем нової чат-кімнати;
- *Метод getAllChats()* – повертає всі об’єкти ChatModel, що знаходяться у базі даних сервісу Chat;
- *Метод addUserToChat()* – записує значення id користувача до об’єкту ChatModel у базі даних сервісу Chat, після долучення користувача до певної чат кімнати.
- *Метод delUserFromChat()* – видаляє значення id користувача з об’єкту ChatModel у базі даних сервісу Chat, після виходу користувача із певної чат кімнати.
- *Метод getChatAdmin()* – повертає значення id адміністратора чат-кімнати з об’єкту ChatModel у базі даних сервісу Chat.

- *Метод getAllChatUsers()* – повертає значення id всіх користувачів чат-кімнати з об’єкту ChatModel у базі даних сервісу Chat.
- *Метод validChatForm()* – перевіряє форму створення нової чат-кімнати.

Функціонал сервісу Message:

- *MessageModel* – клас, що відповідає за створення об’єкту повідомлення для подальшого використання у системі;
- *Метод sendMessage()* – виконує відправлення повідомлення від одного користувача у рамках однієї чат-кімнати;
- *Метод validMessage()* – перевіряє вміст повідомлення за наявності обмежень, створених адміністраторм чат-кімнати, повідомлення з недопустимим вмістом блокуються.
- *Метод saveMessage()* – виконує збереження об’єкту MessageModel у базі даних сервісу Message.
- *Метод hashMessage()* – виконує хешування змісту повідомлення перед збереженням в базу.
- *Метод unhashMessage()* – виконує розхешування змісту повідомлення.
- *Метод getMessagesInChat()* – повертає вміст останніх об’єктів MessageModel у базі даних сервісу Message підчас перегляду історії чату.

Вигляд функціоналу розробленого веб-додатку зі сторони користувача демонструєтьсяна діаграмі прецедентів (рис. 3.4.).

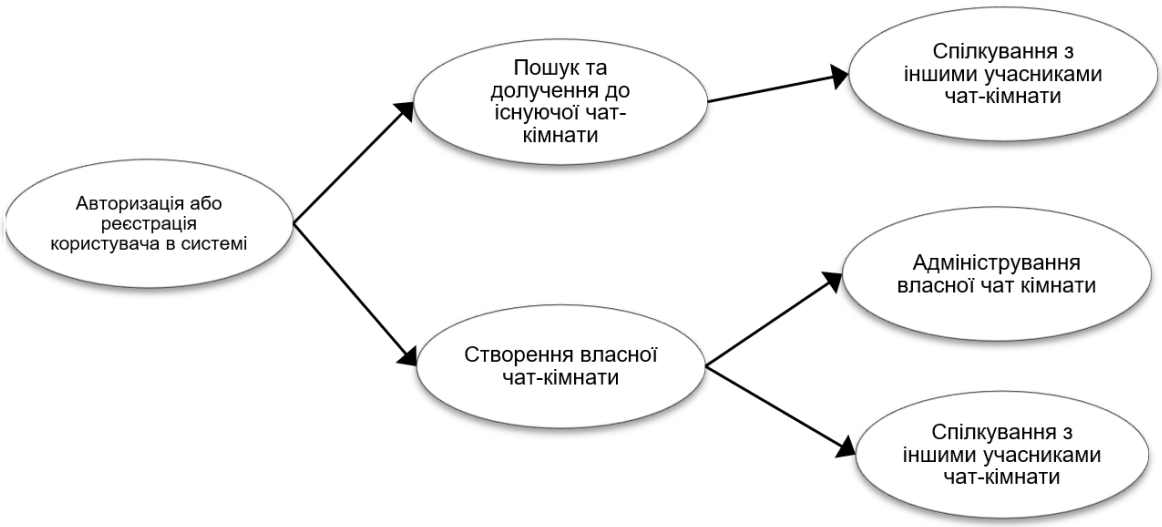


Рис. 3.4. UML діаграма прецедентів

У веб-додатку для створення та адміністрування чат кімнат після авторизації користувач має змогу пошуку та долучення до наявних чат-кімнати, а також створити нову, після чого отримає додатковий функціонал адміністрування власної чат-кімнати.

Опишемо основні сценарії поведінки користувача у системі:

1 Сценарій

Назва сценарію: Реєстрація користувача у системі.

Учасники: Користувач, Система.

Передумови: Відсутність даних користувача в базі даних сервісу User

Результат: Користувач реєструється у системі та може використовувати її функціонал.

Алгоритм сценарію:

1. Користувач заповнює форму реєстрації, а саме 3 поля: ім’я, унікальний логін в системі, пароль;
2. Система перевіряє введені дані
    a. якщо в базі даних сервісу User немає користувача із вказаним логіном – відбувається перехід на 3 крок сценарію;

b. якщо в базі даних сервісу User знаходиться користувач із вказаним логіном, виводиться повідомлення про помилку реєстрації, повернення до 1 кроку сценарію;

3. Виконується хешування введеного користувачем паролю;
4. Введені користувачем дані та хеш паролю зберігаються в базі сервісу User;
5. Користувач отримує повідомлення про успішну реєстрацію в системі;
6. Користувач отримує доступ до функціоналу системи, та можливість виходу із системи.

2 Сценарій

Назва сценарію: Авторизація користувача у системі.

Учасники: Користувач, Система.

Передумови: Наявність даних користувача в базі даних сервісу User

Результат: Користувач авторизується у системі та може використовувати її функціонал.

Алгоритм сценарію:

1. Користувач заповнює форму авторизації, а саме 2 поля: унікальний логін в системі та пароль;
2. Система перевіряє введені дані

a. якщо в базі даних сервісу User немає користувача із вказаним логіном – виводиться повідомлення про помилку авторизації, повернення до 1 кроку сценарію;

b. якщо в базі даних сервісу User знаходиться користувач із вказаним логіном, перевіряється на відповідність пароль:

i. Якщо після хешування пароль відповідає збереженому в базі сервісу User паролю, закріпленим за введеним логіном, відбувається перехід до пункту 3 цього сценарію;

ii. Якщо після хешування пароль НЕ відповідає збереженому в базі сервісу User паролю, закріпленим за введеним логіном – виводиться повідомлення про помилку авторизації, повернення до 1 кроку сценарію;

3. Повідомлення про успішну авторизацію в системі користувача;
4. Користувач може використовувати функціонал системи.

3 Сценарій

Назва: Створення власної чат-кімнати

Учасники: Користувач, Система.

Передумова: Користувач авторизований у системі

Результат: Користувач створює чат-кімнати, в якій може спілкуватися з іншими користувачами та виконувати адміністрування.

Алгоритм сценарію:

1. Користувач натискає на кнопку «Створити нову кімнату» в інтерфейсі системи;
2. Відкривається форма для введення параметрів чат кімнати: назва, тема (опціонально), максимальна кількість учасників;
3. Введені дані разом із автоматично створеним унікальним ідентифікатором кімнати записуються до бази Chat-сервісу, та користувач потрапляє до інтерфейсу чат кімнати, в якому окрім можливості відсилання повідомлень має доступ до кнопки «Адміністрування», яка надає панель адміністрування цієї чат-кімнати.

Детальне розуміння того, як повинен проходити кожен сценарій допомагає виявити та попередити ряд виключних ситуацій, що можуть виникати при взаємодії користувача із системою. В свою чергу, опрацювання таких ситуації розробником, допомагає вберегти сервіси від можливих збоїв, замикань або перенавантажень та підвищити загальну працездатність та відмовостійкість системи.

ВИСНОВКИ ДО РОЗДІЛУ 3

У цьому розділі ми розглянули деталі створення власної системи на основі мікросервісної архітектури. В ході розробки було використано ряд шаблонів проектування, що значно полегшили організацію мікросервісної структури системи. Також розробка виконувалась із дотриманням норм, описаних в попередніх розділах, що повинні забезпечити ефективність системи та необхідну для даного архітектурного стилю відмовостійкість.

Окрім процесу розробки, у розділі також було описано головний функціонал, який система надає користувачу, продемонстровано ієрархію класів кожного сервісу системи. Окрім цього створено ряд можливих сценаріїв взаємодії користувача із системою для обробки виникнення можливих виключних ситуацій.

ВИСНОВКИ

В даній дипломній роботі було проведено дослідження основних принципів побудови мережевих систем на базі мікросервісної архітектури. Виконано аналіз переваг та недоліків даного архітектурного підходу, здійснено порівняння його з класичними архітектурними рішеннями, такими як монолітна тришарова та сервіс-орієнтована архітектура.

Проведено детальний розгляд основних принципів проектування та розгортання мережевих систем виконаних у мікросервісному стилі, основних шаблонів, що використовуються для інтеграції мікросервісів та технології комунікації. Працездатність наведених концепцій продемонстровано на прикладі розробленого веб-додатку. В ході розробки системи було використано ключові компоненти мікросервісних систем, що являють інфраструктурний рівень та надають системі таких властивостей як гнучкість та відмовостійкість. Такими компонентами є: сервіс, що надає єдину точку входу до системи, автоматичний вимикач, сервіс для моніторингу стану активних функціональних сервісів. В рамках розробленого веб-додатку було використано готові реалізації даних шаблонів.

Серед основних переваг мікросервісних систем є їхня гетерогенність, тобто можливість використання незалежних сервісів, кожен з яких використовує свій власний оптимальний стек технологій для реалізації відповідного функціоналу. Для розкриття даної переваги було проведено дослідження базових бібліотек та фреймворків, які надають функціонал створення мікросервісів, на деяких із популярних мов програмування та програмних платформах.

Розглянуто основні питання, що стосуються проблем безпеки взаємодії сервісів у системі та підходів до масштабування мікросервісного додатку. Результатом всіх проведених досліджень є розроблена система яка демонструє працездатність вибраної архітектурної концепції. Більша частина системи

написана мовою програмування Java з використанням її інструментів та бібліотек, що які допомагають в реалізації розподілених мережевих додатків.

У підсумку даної роботи можна з впевненістю стверджувати, що мікросервісна архітектура заслуговує бути розглянутою за основу нового веб-додатка, але в найближчий час не стане найпопулярнішим архітектурним рішенням. Найкраще мікросервіси підійдуть для доповнення чинних монолітних систем, яким для подальшого розвитку необхідний перехід на архітектуру, що дозволяє горизонтальне масштабування. Наразі мікросервісна архітектура показує себе перспективною та з великою ймовірністю буде масово використовуватись у майбутньому.

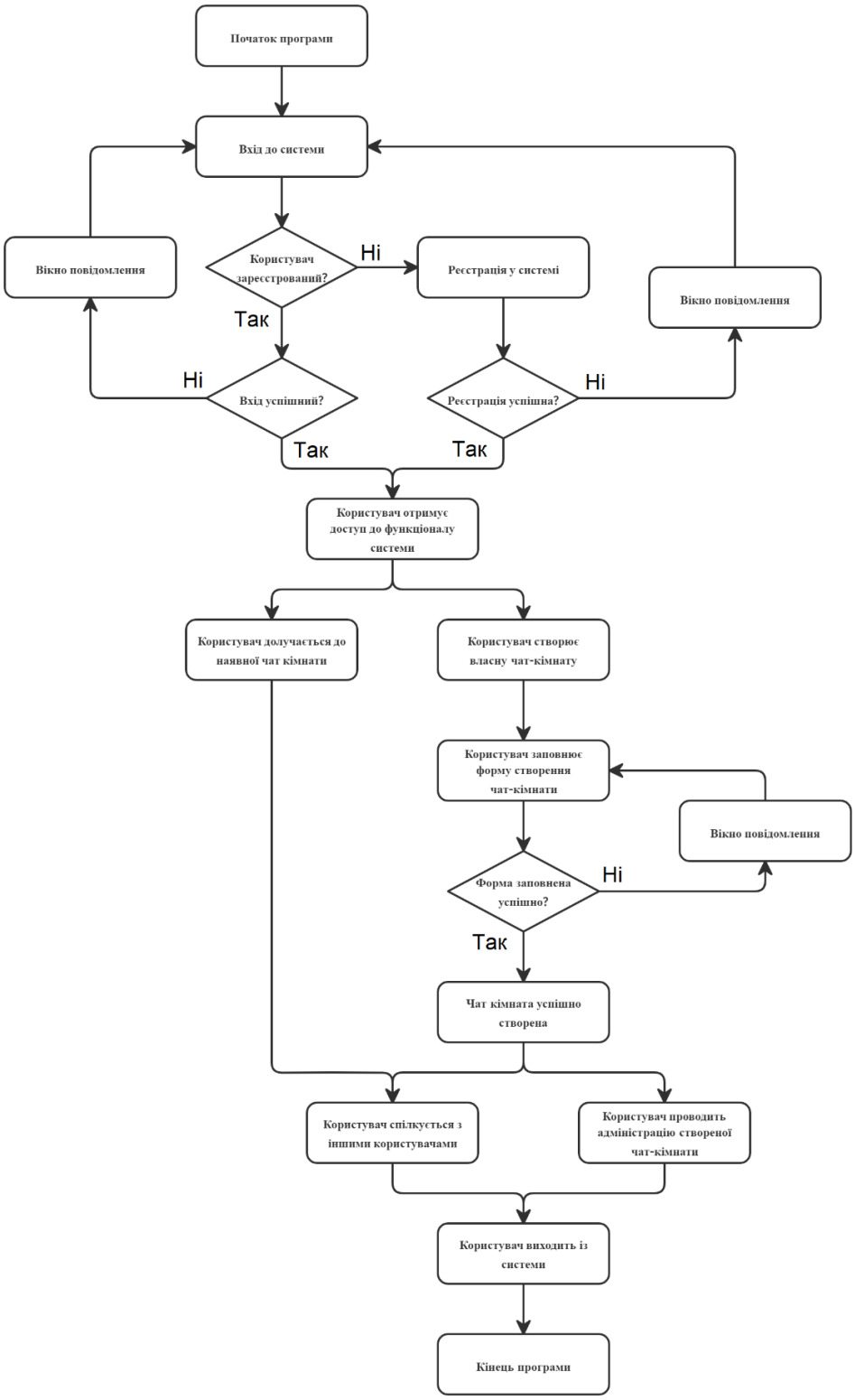
ПЕРЕЛІК ПОСИЛАНЬ

1. Chris Richardson. Pattern: Microservice Architecture [Електронний ресурс] – Режим доступу: http://microservices.io/patterns/microservices.html
2. James Lewis, Martin Fowler Microservices [Електронний ресурс] – Режим доступу: https://martinfowler.com/articles/microservices.html
3. Microsoft – Understanding Service Oriented Architecture [Електронний ресурс] – Режим доступу: https://msdn.microsoft.com/enus/library/aa480021.aspx
4. Leanix, Why Netflix, Amazon, and Apple Care About Microservices [Електронний ресурс] – Режим доступу: https://blog.leanix.net/en/whynetflix-amazon-and-apple-care-about-microservices
5. Ньюмен С. Создание микросервисов. / С. Ньюмен. // Создание микросервисов, Санкт-Петербург: Питер, 2016. – (1). – С. 301–305.
6. Б. Бернс, Паттерны проектирования / Брендан Бернс // Распределенные системы. Паттерны проектирования, СПБ: Питер, 2019. – С. 104–105.
7. Р. Клапчук, В. Харченко, Монолітні веб-сервіси та мікросервіси: порівняння та вибір [Електронний ресурс] – Режим доступу: http://nti.khai.edu:57772/csp/nauchportal/Arhiv/REKS/2017/REKS117/Klapchuk.pdf
8. Ivan Zmerzlyi, Мікросервісна архітектура [Електронний ресурс] – Режим доступу: https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d
9. Accessing API Gateway [Електронний ресурс] – Режим доступу: https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-dg.pdf
10. Офіційний сайт Beanstalkd [Електронний ресурс] – Режим доступу: http://kr.github.io/beanstalkd

11. Офіційний сайт AmazonMQ [Електронний ресурс] – Режим доступу: https://aws.amazon.com
12. Офіційний сайт Spring framework [Електронний ресурс] – Режим доступу: https://spring.io
13. Офіційний сайт Flask [Електронний ресурс] – Режим доступу: http://flask.pocoo.org
14. Офіційний сайт Tornado [Електронний ресурс] – Режим доступу: http://tornadoweb.org
15. Офіційний сайт Nameko [Електронний ресурс] – Режим доступу: https://nameko.readthedocs.io
16. Хорсдал К. Микросервисы на платформе .NET / Хорсдал Кристиан // Микросервисы на платформе .NET, Санкт-Петербург: Питер, 2018. – (3). – С. 40–45.
17. Лонг Дж. Java в облаке. Spring Boot, Spring Cloud, Cloud Foundry / Лонг Джош, Бастани Кеннет // Java в облаке, Санкт-Петербург: Питер, 2018.
18. Core MySQL features [Електронний ресурс] – Режим доступу: https://searchoracle.techtarget.com/definition/MySQL

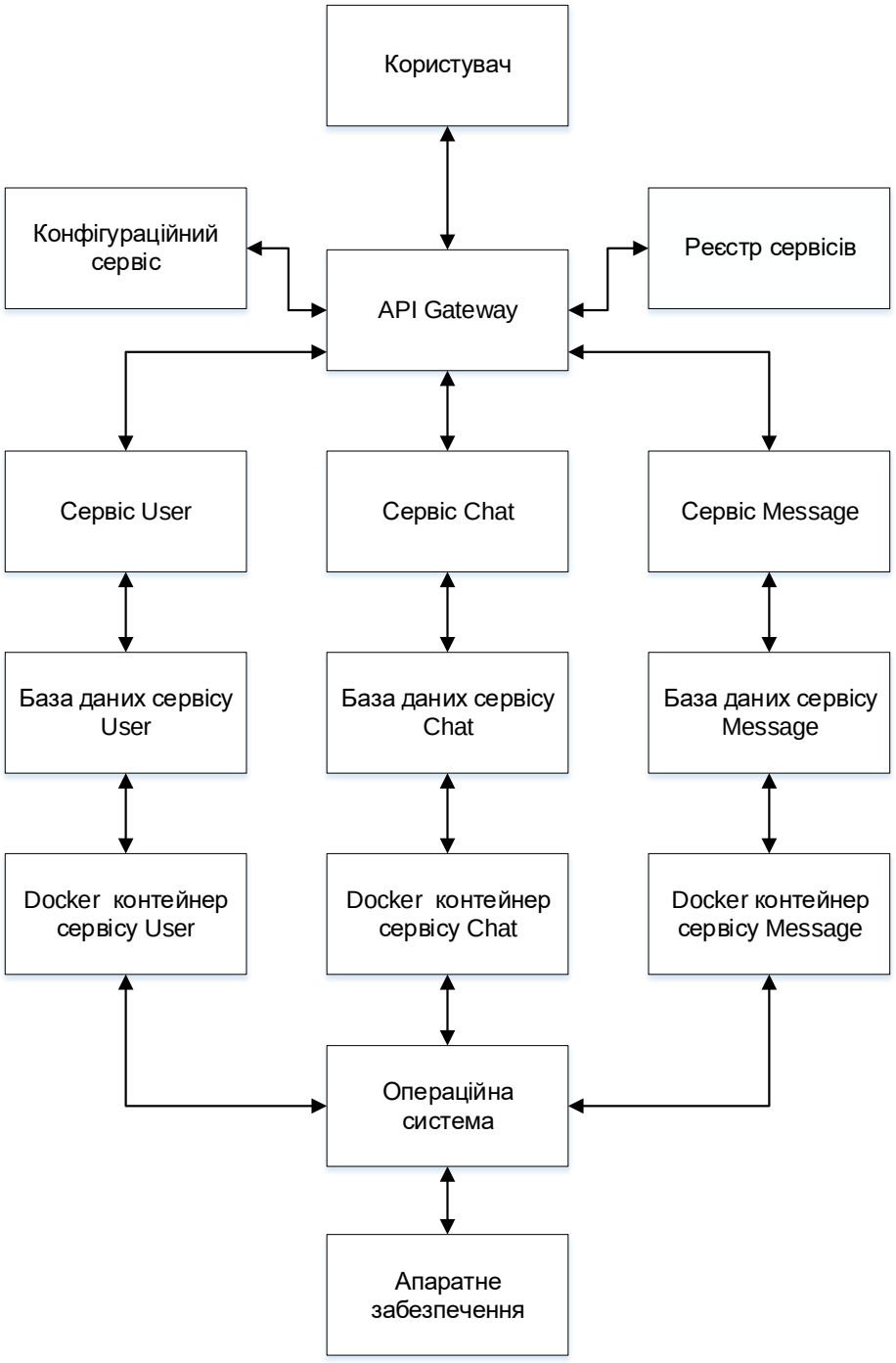
ДОДАТКИ

Схема Алгоритму



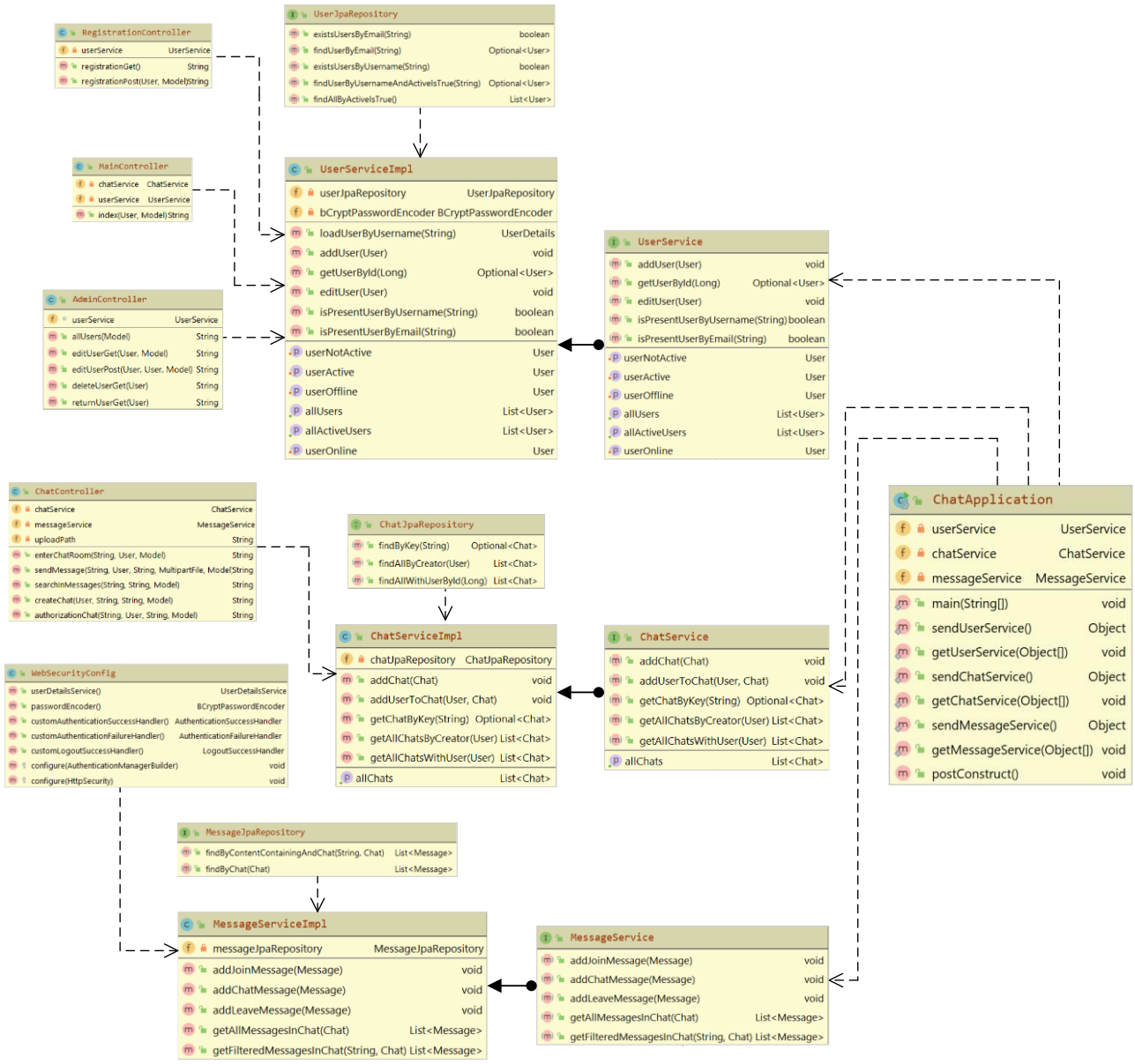
					ДП 6407.04.000 Д1		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		*Дем'янчук П. С.*			*Веб-додаток для створення та адміністрування чат-кімнат на мікросервісній архітектурі* *Додаток А*	Літ.	Аркуш 1 / Аркушів 1
Перевір.		*Каплунов А. В.*					
Н. контр.		*Сімоненко В. П.*				*НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, Група ІО-64*	
Затверд.							

Функціональна Схема



					ДП 6407.05.000 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		*Дем'янчук П. С.*			*Веб-додаток для створення та адміністрування чат-кімнат на мікросервісній архітектурі* *Додаток Б*	Літ.	Аркуш 1 / Аркушів 1
Перевір.		*Каплунов А. В.*					
Н. контр.		*Сімоненко В. П.*				*НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, Група ІО-64*	
Затверд.							

Діграма Класів



					ДП 6407.06.000 ДЗ	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		*Дем'янчук П. С.*			*Веб-додаток для створення та адміністрування чат-кімнат на мікросервісній архітектурі* *Додаток В*	Літ. / Аркуш 1 / Аркушів 1
Перевір.		*Каплунов А. В.*				*НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, Група ІО-64*
Н. контр.		*Сімоненко В. П.*				
Затверд.						

Програмний Код

Клас запуску системи

```
@EnableEurekaServer
@EnableZuulProxy
@SpringBootApplication
public class ChatApplication {
   @Autowired
   private UserService userService;
   private ChatService userService;
   private MessageService messageService;
   public static void main(String[] args) {
      SpringApplication.run(ChatApplication.class, args);
   }
   @PostConstruct
   public void postConstruct() {
       User admin = new User("admin", "admin@admin", "1111",
                                                    Collections.singleton(Role.ADMIN));
      User user = new User("user", "1@1", "1111", new
                                          HashSet<>(Arrays.asList(Role.ADMIN, Role.USER)));
      userService.addUser(admin);
      userService.addUser(user);
   }
}
```

Функціонал User Service

```
@Service
public class ChatServiceImpl implements ChatService {
   @Autowired
   private ChatJpaRepository chatJpaRepository;
   @Transactional
   @Override
   public void addChat(Chat chat) {
      chatJpaRepository.saveAndFlush(chat);
   }
```

```
    @Override
    public void addUserToChat(User user, Chat chat) {
        chat.getUsers().add(user);
        chatJpaRepository.saveAndFlush(chat);
    }
    @Transactional(readOnly = true)
    @Override
    public Optional<Chat> getChatByKey(String key) {
        return chatJpaRepository.findByKey(key);
    }
    @Transactional(readOnly = true)
    @Override
    public List<Chat> getAllChatsByCreator(User user) {
        return chatJpaRepository.findAllByCreator(user);
    }
    @Transactional(readOnly = true)
    @Override
    public List<Chat> getAllChatsWithUser(User user) {
        return chatJpaRepository.findAllWithUserById(user.getId());
    }
    @Transactional(readOnly = true)
    @Override
    public List<Chat> getAllChats() {
        return chatJpaRepository.findAll();
    }
}

@Repository
public interface UserJpaRepository extends JpaRepository<User, Long> {
    boolean existsUsersByEmail(String email);
    Optional<User> findUserByEmail(String email);
    boolean existsUsersByUsername(String username);
    Optional<User> findUserByUsernameAndActiveIsTrue(String username);
    List<User> findAllByActiveIsTrue();
}

@Controller
public class MainController {
    @Autowired
    private ChatService chatService;
    @Autowired
```

```
    private UserService userService;
    @GetMapping("/")
    public String index(@AuthenticationPrincipal User user, Model model) {
        model.addAttribute("username", user.getUsername());
        List<User> allUsers = userService.getAllActiveUsers();
        List<Chat> allChats = chatService.getAllChats();
        List<Chat> allMyChats = chatService.getAllChatsByCreator(user);
        List<Chat> allChatsWithMe = chatService.getAllChatsWithUser(user);
        model.addAttribute("allUsers", allUsers);
        model.addAttribute("allChats", allChats);
        model.addAttribute("allMyChats", allMyChats);
        model.addAttribute("allChatsWithMe", allChatsWithMe);
        return "index";
    }
}

@Controller
@RequestMapping(path = {"/registration"})
public class RegistrationController {
    @Autowired
    private UserService userService;
    @GetMapping
    public String registrationGet() {
        return "registration";
    }
    @PostMapping
    public String registrationPost(User user, Model model) {
        if (userService.isPresentUserByEmail(user.getEmail()) ||
                userService.isPresentUserByEmail(user.getUsername())) {
            model.addAttribute("message", "Such user already exists!");
            return "registration";
        }
        user.setRoles(Collections.singleton(Role.USER));
        userService.addUser(user);
        return "login";
    }
}

@Controller
@PreAuthorize("hasAuthority('ADMIN')")
public class AdminController {
    @Autowire
```

```
    UserService userService;
  @GetMapping("/users")
  public String allUsers(Model model) {
    List<User> allUsers = userService.getAllUsers();
    model.addAttribute("allUsers", allUsers);
    return "users";
  }
  @RequestMapping(path = {"/users/edit/{user}"}, method = RequestMethod.GET)
  public String editUserGet(@PathVariable("user") User user, Model model) {
    user.setPassword(null);
    model.addAttribute("roles", Role.values());
    model.addAttribute("user", user);
    return "user-edit";
  }
  @RequestMapping(path = {"/users/edit/{user}"}, method = RequestMethod.POST)
  public String editUserPost(@ModelAttribute("user") User userToEdit,
                             @PathVariable("user") User user, Model model) {
    if (!(userToEdit.getEmail().equals(user.getEmail())) &&
          userService.isPresentUserByEmail(userToEdit.getEmail())) {
      model.addAttribute("message", "The user with such email already exists!");
      return "user-edit";
    }
    userService.editUser(userToEdit);
    return "redirect:/users";
  }
  @RequestMapping(path = {"users/delete/{user}"}, method = RequestMethod.GET)
  public String deleteUserGet(@PathVariable("user") User user) {
    userService.setUserNotActive(user);
    return "redirect:/users";
  }
  @RequestMapping(path = {"users/return/{user}"}, method = RequestMethod.GET)
  public String returnUserGet(@PathVariable("user") User user) {
    userService.setUserActive(user);
    return "redirect:/users";
  }
}


public class CustomAuthenticationSuccessHandler implements
   AuthenticationSuccessHandler {
  @Autowired
  UserService userService;
```

```
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
                                        HttpServletResponse response,
                                        Authentication authentication)
            throws IOException {
        User user = (User) authentication.getPrincipal();
        userService.setUserOnline(user);
        HttpSession session = request.getSession();
        if (session != null) {
            session.setAttribute("user", user);
        }
        response.sendRedirect("/");
    }
}

public class CustomAuthenticationFailureHandler implements
     AuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest request,
                                        HttpServletResponse response,
                                        AuthenticationException exception)
            throws IOException {
        String errorMessage;
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        if (isNullOrEmpty(username) || isNullOrEmpty(password)) {
            errorMessage = "The form is not fully completed!";
            request.setAttribute("lastEnteredUsername", username);
            request.setAttribute("lastEnteredPassword", password);
        } else if (exception.getClass().isAssignableFrom(BadCredentialsException.class)) {
            errorMessage = "Wrong email or password! Check them and try again.";
            request.setAttribute("lastEnteredUsername", username);
            request.setAttribute("lastEnteredPassword", password);
        } else {
            errorMessage = "Unknown error - " + exception.getMessage();
            request.setAttribute("lastEnteredUsername", username);
            request.setAttribute("lastEnteredPassword", password);
        }
        request.getSession().setAttribute("message", errorMessage);
        response.sendRedirect("/login");
```

```
    }
    private boolean isNullOrEmpty(String requestParameter) {
        return Objects.isNull(requestParameter) || requestParameter.trim().isEmpty();
    }
}
```

Функціонал *Chat Service*

```
@Service
public class ChatServiceImpl implements ChatService {
    @Autowired
    private ChatJpaRepository chatJpaRepository;
    @Transactional
    @Override
    public void addChat(Chat chat) {
        chatJpaRepository.saveAndFlush(chat);
    }
    @Transactional
    @Override
    public void addUserToChat(User user, Chat chat) {
        chat.getUsers().add(user);
        chatJpaRepository.saveAndFlush(chat);
    }
    @Transactional(readOnly = true)
    @Override
    public Optional<Chat> getChatByKey(String key) {
        return chatJpaRepository.findByKey(key);
    }
    @Transactional(readOnly = true)
    @Override
    public List<Chat> getAllChatsByCreator(User user) {
        return chatJpaRepository.findAllByCreator(user);
    }
    @Transactional(readOnly = true)
    @Override
    public List<Chat> getAllChatsWithUser(User user) {
        return chatJpaRepository.findAllWithUserById(user.getId());
    }
    @Transactional(readOnly = true)
    @Override
```

```
    public List<Chat> getAllChats() {
        return chatJpaRepository.findAll();
    }
}

@Repository
public interface ChatJpaRepository extends JpaRepository<Chat, Long> {
    Optional<Chat> findByKey(String key);

    List<Chat> findAllByCreator(User user);

    @Query(value = "SELECT * FROM chat c RIGHT JOIN user_chat uc " +
            "ON c.id = uc.chat_id WHERE uc.user_id = ?1",
            nativeQuery = true)
    List<Chat> findAllWithUserById(Long id);
}
@Controller
@RequestMapping(path = "/chat/room")
public class ChatController {
    @Autowired
    private ChatService chatService;
    @Autowired
    private MessageService messageService;
    @Value("${upload.path}")
    private String uploadPath;
    @GetMapping("/{key}")
    public String enterChatRoom(@PathVariable String key, @AuthenticationPrincipal
     User user, Model model) {
        Optional<Chat> optionalChat = chatService.getChatByKey(key);
        if (optionalChat.isPresent()) {
            Chat chat = optionalChat.get();
            if (!chat.getUsers().contains(user)) {
                String password = chat.getPassword();
                if (Objects.nonNull(password) && !password.equals("")) {
                    model.addAttribute("key", key);
                    return "chat-authorization";
                }
                chatService.addUserToChat(user, chat);
                String joinMessage = String.format("%s is joined to the %s chat!",
     user.getUsername(), key);
 messageService.addJoinMessage(message);
Message message = new Message(joinMessage, user, chat);
```

```
        }
        List<Message> messages = messageService.getAllMessagesInChat(chat);
        model.addAttribute("messages", messages);
        model.addAttribute("username", user.getUsername());
        model.addAttribute("key", key);
        return "chat";
      }
      return "index";
    }
    @PostMapping("/{key}/send/message")
    public String sendMessage(@PathVariable String key,
                              @AuthenticationPrincipal User user,
                              @RequestParam String text,
                              @RequestParam("file") MultipartFile file,
                              Model model) throws IOException {
      Optional<Chat> optionalChat = chatService.getChatByKey(key);
      if (optionalChat.isPresent()) {
        Chat chat = optionalChat.get();
        Message message = new Message(text, user, chat);
        if (Objects.nonNull(file) &&!(file.getOriginalFilename().isEmpty())) {
          File uploadDir = new File(uploadPath);

          if (!uploadDir.exists()) {
            uploadDir.mkdir();
          }
          String uuidFile = UUID.randomUUID().toString();
          String[] filePath = file.getOriginalFilename().split("\\\\");
          String resultFilename = uuidFile + "." + filePath[filePath.length - 1];

          file.transferTo(new File(String.format("%s/%s", uploadPath, resultFilename)));
          message.setFilename(resultFilename);
        }
        messageService.addChatMessage(message);
      }
      return String.format("redirect:/chat/room/%s", key);
    }
    @PostMapping("/{key}/messages/search")
    public String searchInMessages(@PathVariable String key, @RequestParam String
     text, Model model) {
      Optional<Chat> optionalChat = chatService.getChatByKey(key);
      if (optionalChat.isPresent()) {
        Chat chat = optionalChat.get();
```

```
            List<Message> messages = messageService.getFilteredMessagesInChat(text,
        chat);
            model.addAttribute("messages", messages);
            model.addAttribute("key", key);
        }
        return "chat";
    }
    @PostMapping("/create")
    public String createChat(@AuthenticationPrincipal User user,
                             @RequestParam String chatKey,
                             @RequestParam() String chatPassword, Model model) {
        Chat chat = new Chat(chatKey, user);
        if (Objects.nonNull(chatPassword) && !chatPassword.equals("")) {
            chat.setPassword(chatPassword);
        }
        chatService.addChat(chat);
        return String.format("redirect:/chat/room/%s", chatKey);
    }
    @PostMapping("/{key}/authorization")
    public String authorizationChat(@PathVariable String key, @AuthenticationPrincipal
        User user, @RequestParam String chatPassword, Model model) {
        Optional<Chat> optionalChat = chatService.getChatByKey(key);
        if (optionalChat.isPresent()) {
            Chat chat = optionalChat.get();
            if (chatPassword.equals(chat.getPassword())) {
                chatService.addUserToChat(user, chat);
                String joinMessage = String.format("%s is joined to the %s chat!",
        user.getUsername(), key);
                Message message = new Message(joinMessage, user, chat);
                messageService.addJoinMessage(message);
                return String.format("redirect:/chat/room/%s", key);
            } else {
                model.addAttribute("message", "You entered the wrong password!");
                return "chat-authorization";
            }
        }
        return "index";
    }
}
```

Функціонал *Message Service*

```
@Service
public class MessageServiceImpl implements MessageService {
    @Autowired
    private MessageJpaRepository messageJpaRepository;
    @Transactional
    @Override
    public void addJoinMessage(Message message) {
        message.setType(MessageType.JOIN);
        messageJpaRepository.saveAndFlush(message);
    }
    @Transactional
    @Override
    public void addChatMessage(Message message) {
        message.setType(MessageType.CHAT);
        messageJpaRepository.saveAndFlush(message);
    }
    @Transactional
    @Override
    public void addLeaveMessage(Message message) {
        message.setType(MessageType.LEAVE);
        messageJpaRepository.saveAndFlush(message);
    }
    @Transactional(readOnly = true)
    @Override
    public List<Message> getAllMessagesInChat(Chat chat) {
        return messageJpaRepository.findByChat(chat);
    }
    @Transactional(readOnly = true)
    @Override
    public List<Message> getFilteredMessagesInChat(String text, Chat chat) {
        return messageJpaRepository.findByContentContainingAndChat(text, chat);
    }
}
```

```
@Repository
public interface UserJpaRepository extends JpaRepository<User, Long> {
   boolean existsUsersByEmail(String email);
   Optional<User> findUserByEmail(String email);
  boolean existsUsersByUsername(String username);
   Optional<User> findUserByUsernameAndActiveIsTrue(String username);
   List<User> findAllByActiveIsTrue();
}

@Configuration
public class WebConfig implements WebMvcConfigurer {
   @Value("${upload.path}")
   private String uploadPath;
   @Override
   public void addResourceHandlers(ResourceHandlerRegistry registry) {
      registry.addResourceHandler("/img/**")
            .addResourceLocations(String.format("file:/%s/", uploadPath));
      registry.addResourceHandler("/static/uploads/**")
            .addResourceLocations("classpath:/static/uploads/");
   }
   public void addViewControllers(ViewControllerRegistry registry) {
      registry.addViewController("/login").setViewName("login");
   }
   @Bean
   public LayoutDialect layoutDialect() {
      return new LayoutDialect();
   }
}
```